

COMPUTE!'s Guide to



TI-99/4A Sound and Graphics

Raymond J. Herold

A complete guide to sound, graphics, and speech synthesis on the TI-99/4A, including arcade-style games, music routines, and educational programs, ready to type in and use.

A **COMPUTE! Books** Publication
\$12.95

COMPUTE!'s Guide to
TI-99/4A
SOUND
AND
GRAPHICS

Raymond J. Herold

COMPUTE! Publications, Inc. 
One of the ABC Publishing Companies

Greensboro, North Carolina

TI-99/4A is a registered trademark of Texas Instruments, Inc.

Copyright 1984, COMPUTE! Publications, Inc. All rights reserved.

Reproduction or translation of any part of this work beyond that permitted by Sections 107 and 108 of the United States Copyright Act without the permission of the copyright owner is unlawful.

Printed in the United States of America

ISBN 0-942386-46-9

10 9 8 7 6 5 4 3 2 1

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is one of the ABC Publishing Companies and is not associated with any manufacturer of personal computers. TI-99/4A is a trademark of Texas Instruments.

Contents

| | |
|--|----|
| Foreword | v |
| Chapter 1: Sound and Graphics | 1 |
| Graphics | 3 |
| Sound | 4 |
| Speech Synthesis | 5 |
| Putting the Pieces Together | 5 |
| What You'll Need | 6 |
| Chapter 2: Introduction to Graphics | 7 |
| TI Graphics | 10 |
| TI Screen Characteristics | 10 |
| Character Definition | 14 |
| Defining Your Own Characters | 18 |
| Using DISPLAY AT | 24 |
| Changing Patterns | 24 |
| Color | 25 |
| Bars and Lines | 27 |
| Diagonals | 30 |
| Character Concatenation | 33 |
| Special Effects | 35 |
| High-Resolution Graphics | 40 |
| Chapter 3: Sprites | 49 |
| The TMS9918A Video Display Processor | 52 |
| VDP RAM | 54 |
| Putting It on the Screen | 54 |
| Defining Sprites | 56 |
| CALL Sprite Subprogram Examples | 59 |
| Sprite Sizes | 61 |
| Deleting Sprites | 66 |
| Sprite Color | 66 |
| Controlling Sprites in BASIC Programs | 69 |
| Changing a Sprite's Location | 69 |
| Changing a Sprite's Motion | 70 |
| Controlling Sprites with Joysticks | 71 |
| Controlling Sprites from the Keyboard | 72 |
| Joystick and Keyboard Control | 75 |
| Changing Sprite Patterns | 76 |
| Combining Techniques | 77 |
| Sprite Editor | 80 |

| | |
|---|----------------|
| Chapter 4: Advanced Sprite-Handling Techniques . . | 85 |
| Finding a Sprite's Screen Position | 87 |
| Determining When Sprites Are Coincidental | 93 |
| Determining Sprite Distances | 97 |
| Factors Affecting POSITION, COINC, and DISTANCE . | 102 |
| Writing a Graphics Program | 103 |
| The Scenario | 103 |
| Defining the Graphics | 104 |
| Controlling the Action | 108 |
| Chapter 5: Sound | 113 |
| What Is Sound? | 115 |
| Making Sound on the TI | 115 |
| Turning Sound into Music | 117 |
| Basic Notation | 117 |
| Making Music on the TI | 126 |
| Sound Effects | 143 |
| Rocket in Motion | 144 |
| Morse Code | 144 |
| Computer | 144 |
| Sirens | 145 |
| Bomb and Explosion | 145 |
| Bells | 146 |
| Chapter 6: Speech Synthesis | 147 |
| Expanding the Resident Vocabulary | 155 |
| The Text-to-Speech Diskette | 161 |
| Chapter 7: Putting It All Together | 173 |
| Mimic | 175 |
| Shooting Gallery | 179 |
| Alphabet Invasion | 184 |
| Banzai Bunny | 190 |
| Zone Defender | 195 |
| Addition Climber | 198 |
| Slot Machine | 205 |
| Index | 209 |

Foreword

If you're like most TI computer users, you've probably wondered how to use its sophisticated graphics and sound in programs of your own. Now, at last, here's a book that shows you exactly how.

COMPUTE!'s Guide to TI-99/4A Sound and Graphics explores your computer's sound and graphics capabilities and puts them into words that every TI user can understand. From simple graphics to complex speech synthesis, every aspect is thoroughly covered with step-by-step explanations. Clear, concise examples show you how specific sound and graphics commands are used, and longer programs demonstrate how those individual commands can be combined into exciting and captivating programs.

As you use this guide, you'll discover a great deal about how your TI works. You'll see how displays are produced on the screen. You'll learn how graphics characters are created and controlled. You'll explore sound effects, music, and even speech synthesis.

To show how sound and graphics techniques can actually be used, a number of full-scale programs have also been included. These range from "Alphabet Invasion," which challenges you to unscramble letters after they are beamed down by an alien spaceship, to "Mouse Maze," a joystick-controlled version of cat-and-mouse. "Addition Climber" combines sprite graphics and speech synthesis to form a fascinating educational game for young children, while "Zone Defender" is an arcade-style action game that rivals the best for excitement and appeal.

Whether you're an advanced programmer or a beginning computer hobbyist, you'll find this an extremely helpful book. With it—and your own creativity—you'll be ready to explore the exciting world of TI sound and graphics.



1

Sound and Graphics

Sound and Graphics

COMPUTE!'s *Guide to TI-99/4A Sound and Graphics* is written for anyone who owns, or plans to own, the Texas Instruments TI-99/4A home computer. It is an extremely powerful machine, in spite of its small size, and it offers a great many sophisticated features for both beginning and advanced programmers.

But it takes more than features to make a great computer system. The key to getting more from any home computer is to understand how its features work and how you can use them in your programs.

This book provides you with a step-by-step guide to using three of the most exciting features that your TI has to offer: graphics, sound, and speech synthesis. The engineers at Texas Instruments went to great lengths to place them at your disposal, and there is no better way to enhance a program than by using one or more of them in your programs.

This book is geared toward users of Extended BASIC. Available as a plug-in command module which fits into the TI console, Extended BASIC is much more powerful than the standard BASIC that's built-in. There is some overlap between the two versions of the language. But since Extended BASIC has far greater capabilities, it is better suited to most of the techniques and concepts in this book.

If you're new to programming, you'll find it fun to discover the many things that your TI can do. But even if you're an experienced programmer, you can expect a surprise or two as you work with this guide.

Graphics

There are very few programs that would not benefit from the addition of graphics. But in spite of the obvious benefits, many programmers hesitate to include graphics in their programs. They fear that the techniques involved are tricky and esoteric. But programming graphics on the TI is not that complicated. In fact, it can be a lot of fun.

Chapter 2 introduces you to the concepts and terminology of graphics programming and to nonsprite graphics techniques. In addition, it includes a short discussion of high-

resolution graphics, providing the necessary foundation for subsequent discussions of sprites.

Sprites are graphics characters which, once defined, can be manipulated on the screen in a variety of ways. They offer a great many advantages over nonsprite graphics. Chapters 3 and 4 explain how these unique graphics characters are defined, displayed, and controlled.

These chapters are presented in a how-to fashion, progressing from basic concepts and terminology to the use of advanced techniques. Numerous miniprograms are included to illustrate various programming techniques, and all of these programs are fully explained.

Since learning is best accomplished by doing, you should type in and try the miniprograms. You'll gain a much greater understanding of how graphics techniques work if you can see them in operation.

In addition to the miniprograms, every chapter includes at least one much larger program. Each such program uses a combination of the techniques discussed in the chapter and is accompanied by a detailed explanation of how it works.

Once you have gone through all the graphics chapters, you should be well on your way to using many of the techniques in original programs of your own. But remember that the graphics section covers quite a bit of material. If you start skipping around, you are likely to miss some critical point, so it's a good idea to work through the chapters in order.

Sound

Another useful feature that your TI has to offer is sound generation, which is thoroughly examined in Chapter 5. That chapter considers sound primarily as a program enhancement device. An educational program, for instance, might include a routine to reward a correct answer with a short musical passage. Sound has obvious value in games, too. In a space game, isn't it much more dramatic to hear, as well as see, your spaceship fire its lasers? These are just two of the many cases where your computer's ability to produce sound can add to a program's effectiveness.

This chapter also shows you how to make music on your TI. You don't know very much about music? That's OK; the basics will all be explained.

You'll find several complete programs in this chapter too,

including one that helps you learn the names of the notes on a musical staff. Another program demonstrates your TI's artistic abilities by playing a Bach prelude—in three parts!

Speech Synthesis

The third major feature discussed in this book is speech synthesis. The ability to generate speech is a relatively new capability for the home computer, and the extra dimension that it adds opens up some intriguing possibilities.

Currently, speech synthesis is found primarily in educational programs and in an occasional game. Its use in those areas will certainly increase, but there are other applications where it may have an even greater impact. For example, you hear much these days about so-called "user-friendly" computers. Although this phrase has been greatly overused, it takes on special significance when you consider speech synthesis. After all, what could be more friendly than a computer that literally tells you what to do? Imagine a computer that reminds a businessman of his appointments, not by listing them on a monitor, but by actually speaking to him. How about a home computer that reads to the disabled or blind? Why not have the ever-patient computer teach Junior how to spell? Speech synthesis technology has made all of these things possible.

As mentioned earlier, the main focus of this book is on concepts and techniques as they apply to Extended BASIC. This provides us with two approaches to using the speech synthesizer. One uses Extended BASIC's resident vocabulary, while the other takes advantage of the text-to-speech disk software sold by Texas Instruments. Chapter 6 examines both and includes several programs which illustrate the concepts and techniques described.

Putting the Pieces Together

The final chapter consists of seven complete programs, which combine the features and techniques discussed in the rest of the book, along with a detailed explanation of how the parts of each program fit together.

As you work through the chapters, you will notice that most of the program examples are games. Games present one of the most obvious uses of the graphics, sound, and speech synthesis capabilities of your computer. Though recreation is a

major reason to purchase a home computer like the TI, other applications can also benefit from the use of the graphics, sound, and speech synthesis. Several nongame programs are included to illustrate such applications.

What You'll Need

To effectively use this book, you will need a TI-99/4A computer, the Extended BASIC command module, and at least a black-and-white or monochrome monitor. In addition, the following items will be helpful:

| | |
|--|--|
| Color monitor or TV | Most of the programs deal with multicolor graphics. While they can still be run with a black-and-white TV or a monochrome monitor, the effect will be much better in color. |
| Joysticks | Several of the game programs use these to control the action. |
| Speech synthesis | Chapter 6 is devoted to TI's speech synthesizer module. |
| Text-to-speech | There are two ways to produce speech on your TI. One uses the Extended BASIC resident vocabulary; the other uses this software package. |
| Peripheral expansion box, disk controller card, and disk drive | Disks provide an efficient, convenient way to save and reuse the programs in this book. Of course, you will have to have a disk drive to use the text-to-speech disk software. |

Introduction to Graphics

2

Introduction to Graphics

Computer graphics can be thought of simply as pictures drawn on a TV or monitor. They can consist of anything from a simple stick man to a complex three-dimensional object, and you will find them used in applications ranging from games and education to business, science, and engineering.

Computer graphics have been around for quite some time, although not with the degree of sophistication available today. They first appeared in the mid-1950s, when display devices were limited to oscilloscopes or relatively primitive black-and-white television-type CRTs.

Like other aspects of computer science, graphics technology has evolved dramatically over the years. As a result, a new vocabulary has been developed to describe various aspects of the field. Most of the new terms don't apply to home computers like the TI. But those that do are defined here:

- ASCII** American Standard Code for Information Interchange. This is a standardized code developed by the American National Standards Institute (ANSI). It defines a character set (letters, numbers, special symbols) which can be used by a wide variety of computers. This gives these computers a common thread for talking to each other or for using the same peripherals. Your TI uses the ASCII character set.
- Pixel** A single picture element. Graphics images on a TV or monitor are comprised of numerous dots which glow to form the image. The pixel is the smallest dot that a computer can illuminate.
- Resolution** The relative clarity and quality of an image displayed on a TV set or monitor. It is expressed as a matrix (for example, 256 x 192), usually in pixels, which identifies the number of columns and rows available on the screen. If you think of the screen as a piece of graph paper, you can see that the larger the number of pixels, the better the resolution.

TI Graphics

The TI-99/4A has four different modes for displaying characters on the screen. They are called *graphics mode*, *multicolor mode*, *text mode*, and *bitmapped mode*. Each serves a particular purpose.

Graphics mode lets you display the full ASCII character set as well as define your own characters. Characters are defined by an 8 x 8 matrix, and each character can use any two of the 16 available colors. Graphics mode also gives you access to TI sprites. This is the only mode available from BASIC.

Multicolor mode provides an easier method for drawing pictures than does graphics mode. It defines each character on the screen as a 4 x 4 matrix rather than as the standard 8 x 8 matrix. Each character can be assigned a color. This mode also allows you to use sprites, but it cannot access the ASCII character set.

Text mode divides the screen into 40 columns, rather than 32, by defining each character in a 6 x 8 matrix. It is limited to one background (screen) color and one dot (character) color. It does not support sprites. Text mode is most useful in word processing applications. It is used by the Editor/Assembler and TI-Writer.

Bitmap is the highest resolution mode available on the TI. It allows you to define two colors for each 8 x 1 pixel group. This would allow a single 8 x 8 character to contain all 16 colors. Although sprites may be used in this mode, the automatic motion features are not available. This mode is used in games such as PARSEC and is accessible from Assembler Language.

This book will center on graphics mode, since it is the only one available from BASIC. The other modes were noted for background information; for additional information concerning them, you should consult the TI Editor/Assembler manual.

TI Screen Characteristics

The TI-99/4A screen measures 32 character columns by 24 character rows (Figure 2-1). This gives a total of 768 possible character locations. A particular character position on the screen is defined by its row and column. For example, the character in the first column of row three is located at row 3, column 1.

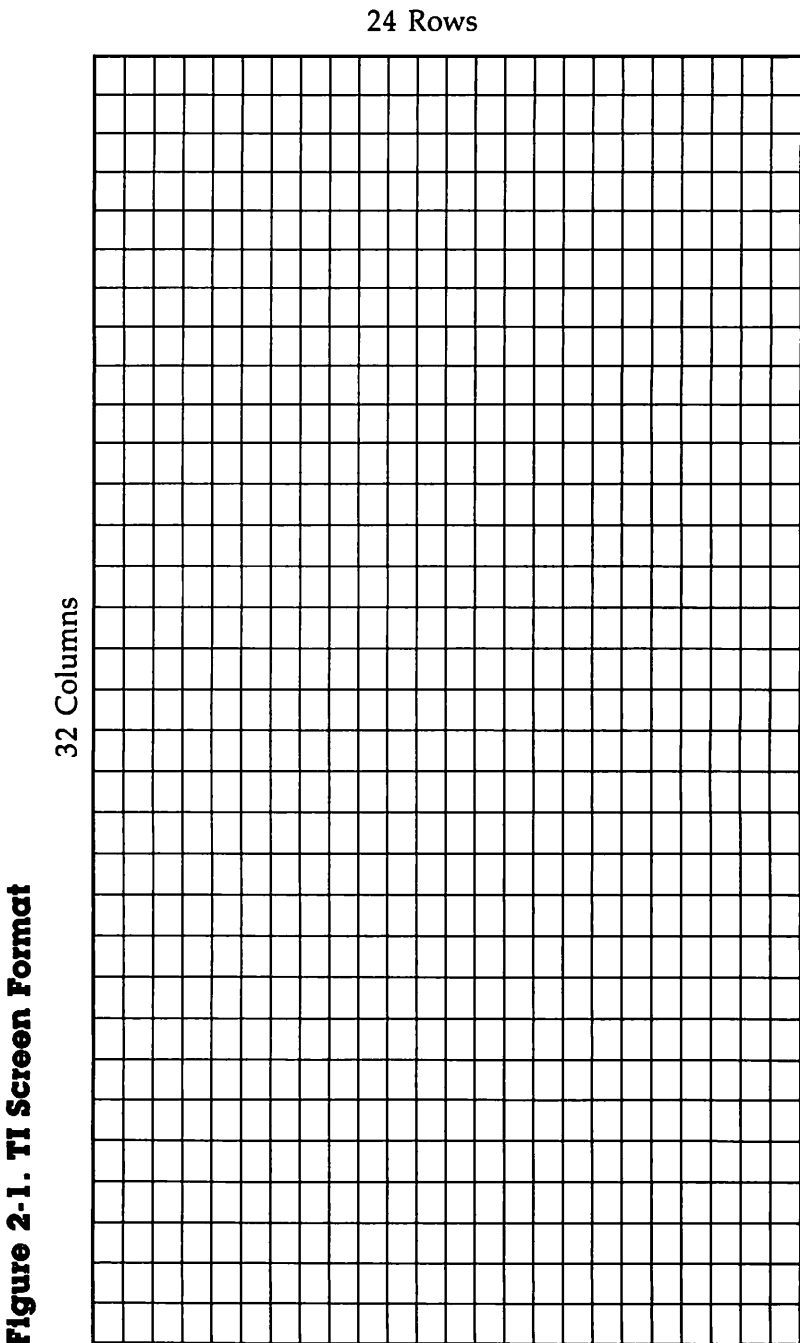
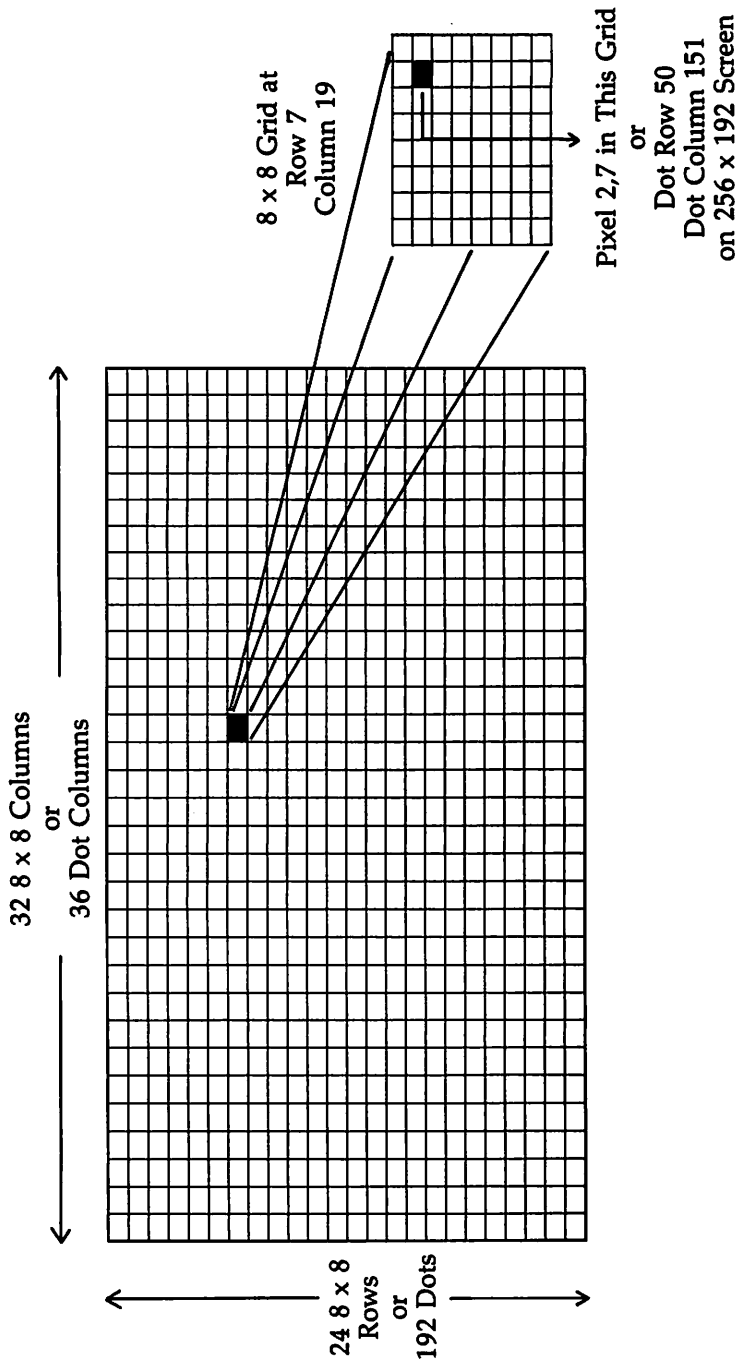


Figure 2-1. TI Screen Format

Figure 2-2. TI Screen Dot-Rows and Dot-Columns



Each of the 768 screen positions is made up of an 8 x 8 matrix, or *grid* (Figure 2-2), which results in a screen measuring 256 pixels (dot-columns) by 192 pixels (dot-rows). Simple multiplication shows that the screen contains 49,152 pixels:

8 pixels x 8 pixels = 64 pixels per character

64 pixels per character x 768 characters = 49,152 pixels

256 columns x 192 rows = 49,152 pixels

That's a lot of pixels. The thing to keep in mind, however, is that BASIC cannot access each of those pixels directly. Instead, pixels must be accessed indirectly by determining the character position to which they belong. For example, if you wanted to turn on the pixel at dot-row 50 and dot-column 151, you would first have to determine the location of that pixel within the 32 x 24 character grid. You could do this with the following BASIC routine:

```
100 WORK=DOTROW/8
110 IF INT(WORK)=WORK THEN 150
120 CHARROW=INT(WORK)+1
130 BITROW=(WORK-INT(WORK))*8
140 GOTO 170
150 CHARROW=WORK
160 BITROW=8
170 .
180 .
190 .
```

Line 100 divides the number of dot-rows (50 in our example) by the 8 dot-rows contained in each character-row. Line 110 determines whether or not the result of that division is an integer. If it is, the routine branches to line 150. If the result has a fractional remainder, the routine proceeds to line 120 where the character-row is determined by taking the integer portion of the number and adding 1 to it.

Line 130 determines the bit-row of the pixel within the 8 x 8 grid by calculating the fractional portion of the character-row and multiplying it by 8 (bit-rows per character-row). Line 140 branches to the end of the routine, since the desired information has already been calculated. Line 150 is processed if the result of the division in line 100 is an integer. It simply assigns the integer result to CHARROW. Line 160 sets the bit-row of the pixel to 8. In this case, no fractional portion of a

character-row is involved; thus, the pixel must be located in the last bit-row of this character-row.

The same procedure can be used to determine the character-column and the bit-column. Using the pixel in our example would result in the following:

```
WORK=DOTROW/B
6.25=50/8
CHARROW=INT(WORK)+1
7=6+1
BITROW=(WORK-INT(WORK))*8
2=0.25*8

WORK=DOTCOL/8
18.875=151/8
CHARCOL=INT(WORK)+1
19=18+1
BITCOL=(WORK-INT(WORK))*8
7=0.875*8
```

Thus, that particular pixel resides at bit-row 2, bit-column 7 within the 8 x 8 character at character-row 7, character-column 19. You will soon see the importance of such information.

Character Definition

To display a character on the 32 x 24 screen, the TI has to know the *pattern* for that character. The pattern is drawn on an 8 x 8 grid. The individual dots, or *bits*, which are *on* within the grid, determine what the character will look like on the screen.

The TI has a set of these patterns built into ROM (Read Only Memory), and the pattern used by a particular character is determined by its ASCII character code (Table 2-1). For example, you can see from the table that ASCII character code 65 designates the bit pattern for the letter A. The actual pattern in the 8 x 8 grid looks like this:

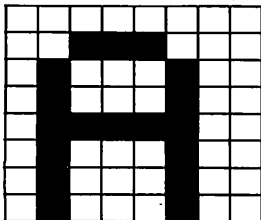
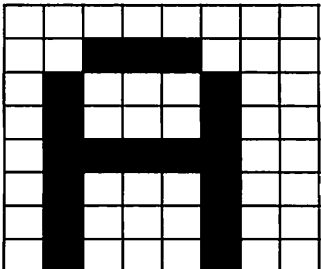


Table 2-1. ASCII Character Codes

| ASCII | Character | ASCII | Character |
|-------|-----------------------|---------|-------------------------|
| 30 | (cursor) | 67 | C |
| 31 | (edge character) | 68 | D |
| 32 | (space) | 69 | E |
| 33 | ! (exclamation point) | 70 | F |
| 34 | " (quote) | 71 | G |
| 35 | # (number sign) | 72 | H |
| 36 | \$ (dollar) | 73 | I |
| 37 | % (percent) | 74 | J |
| 38 | & (ampersand) | 75 | K |
| 39 | ' (apostrophe) | 76 | L |
| 40 | ((open parenthesis) | 77 | M |
| 41 |) (close parenthesis) | 78 | N |
| 42 | * (asterisk) | 79 | O |
| 43 | + (plus) | 80 | P |
| 44 | , (comma) | 81 | Q |
| 45 | - (minus) | 82 | R |
| 46 | . (period) | 83 | S |
| 47 | / (slash) | 84 | T |
| 48 | 0 | 85 | U |
| 49 | 1 | 86 | V |
| 50 | 2 | 87 | W |
| 51 | 3 | 88 | X |
| 52 | 4 | 89 | Y |
| 53 | 5 | 90 | Z |
| 54 | 6 | 91 | [(open bracket) |
| 55 | 7 | 92 | (reverse slash) |
| 56 | 8 | 93 |] (close bracket) |
| 57 | 9 | 94 | (exponentiation) |
| 58 | : (colon) | 95 | (underline) |
| 59 | ; (semicolon) | 96 | |
| 60 | < (less than) | 97-122 | (lowercase letters a-z) |
| 61 | = (equals) | 123 | { (open brace) |
| 62 | > (greater than) | 124 | |
| 63 | ? (question mark) | 125 | } (close brace) |
| 64 | @ (at sign) | 126 | |
| 65 | A | 127 | DEL |
| 66 | B | 128-143 | (see note) |

Note: ASCII codes 128-143 are undefined in normal operation. They are, however, available to Extended BASIC programs.

But how is such a pattern defined for the computer? The first step is to break the 8 x 8 grid into two 4 x 8 sections. The *off* bits (those not appearing on the screen) are assigned a value of 0. The *on* bits (those that will be illuminated to form the character) are assigned a value of 1. Then, working left to right and top to bottom, the pattern is broken into groups of four bits each. This results in 16 groups of 4 bits each (64 bits total) and looks like this:

| | |
|---|-------------------|
|  | Bit Values |
| = | 0000 0000 |
| = | 0011 1000 |
| = | 0100 0100 |
| = | 0100 0100 |
| = | 0111 1100 |
| = | 0100 0100 |
| = | 0100 0100 |
| = | 0100 0100 |

Combining all 16 groups, the bit pattern constitutes a *binary* number that looks like this:

```
000000000011100001000100010001111100010001000
100010001000100
```

You may think that this is an awkward way to describe a pattern, and you're right. Fortunately, there is a better way. This 64-digit number can be reduced to just 16 digits by using *hexadecimal* notation.

The digits of decimal numbers are based on the powers of 10 ($10^0=1$, $10^1=10$, $10^2=100$, $10^3=1000$, and so on). Hexadecimal numbers, on the other hand, use the *base 16* number system. Each digit in a hexadecimal number is based on a power of 16 ($16^0=1$, $16^1=16$, $16^2=256$, $16^3=4096$, and so on).

If you think about it, you'll see that hexadecimal numbers need more than ten symbols to represent the individual digits. Try to imagine a 16-fingered Martian who regularly uses the hexadecimal number system. If he were to count his fingers, he would have to use a single-digit name for each of the first fifteen fingers before he could count his sixteenth finger as "10" ($(1 \times 16) + (0 \times 1)$).

The letters A through F represent the additional digits

required by hexadecimal numbers. As a result, counting from 1 to 16 would go like this:

Decimal: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
 Hex: 1 2 3 4 5 6 7 8 9 A B C D E F 10

What does this have to do with character patterns?
 Remember that there were 16 four-digit binary numbers in the pattern. As it turns out, any combination of four binary digits can be represented by one hexadecimal digit ($2^4=16$). For example, the binary number (or bit pattern) 1101 equals D in hexadecimal, as shown:

Powers of 2: 8 4 2 1
 Bit pattern: 1 1 0 1

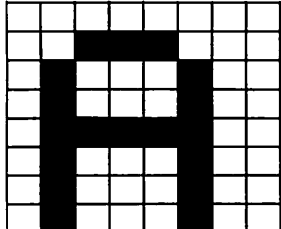
equals

$$\begin{array}{r} 1 \times 8 = 8 \\ 1 \times 4 = 4 \\ 0 \times 2 = 0 \\ 1 \times 1 = 1 \\ \hline 13 = D \text{ in hex} \end{array}$$

If you aren't comfortable with the math, you can use the following table:

| Binary Hex | Binary Hex |
|------------|------------|
| 0000 = 0 | 1000 = 8 |
| 0001 = 1 | 1001 = 9 |
| 0010 = 2 | 1010 = A |
| 0011 = 3 | 1011 = B |
| 0100 = 4 | 1100 = C |
| 0101 = 5 | 1101 = D |
| 0110 = 6 | 1110 = E |
| 0111 = 7 | 1111 = F |

By looking at the grid one more time, you can now see that the pattern for the letter A is 003844447C444444:

| | Binary | Hex |
|---|-------------|-------|
|  | = 0000 0000 | = 0 0 |
| | = 0011 1000 | = 3 8 |
| | = 0100 0100 | = 4 4 |
| | = 0100 0100 | = 4 4 |
| | = 0111 1100 | = 7 C |
| | = 0100 0100 | = 4 4 |
| | = 0100 0100 | = 4 4 |
| | = 0100 0100 | = 4 4 |

Extended BASIC has an instruction (CALL CHARPAT) that will return the hexadecimal pattern for a particular ASCII character. You can type in and RUN the following program to confirm the pattern for an A:

```
100 CALL CLEAR
110 CALL CHARPAT(65,PAT$)
120 PRINT PAT$
130 STOP
```

Defining Your Own Characters

If you were limited to the standard ASCII character set, you wouldn't have much flexibility in producing graphics on the screen. Luckily, the TI gives you a way to define your own characters: the CALL CHAR statement. It lets you change the pattern for any ASCII character code and is, in effect, the opposite of the CALL CHARPAT statement. It has the following format:

CALL CHAR (character code,pattern identifier)

Character code is the ASCII character code in which you will place the new pattern. It can be either a numeric value in the range 32–143 or a *numeric variable* which contains a value of 32–143.

Pattern identifier is the hexadecimal pattern describing what the new character will look like. It consists of a hexadecimal number 0 to 64 digits long, or a *string variable* which contains a hex number 0 to 64 characters long.

Ordinarily, it takes 16 hex digits to define a character. If the hex number you provide is less than 16 digits long, the computer assumes the rest of the digits are 0. For example, the computer would translate the number 005C11FF1212 into 005C11FF12120000. If the number is between 17 and 32 digits in length, then two consecutive ASCII character codes are defined starting with the one indicated by *character code*. A hex number 33 to 48 digits long defines three consecutive ASCII character codes; a number 49 to 64 digits long defines four.

The following examples show various ways of using the CALL CHAR statement:

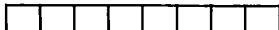
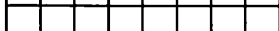
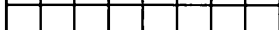


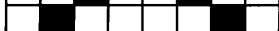
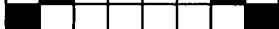
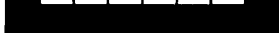
```
100 CALL CHAR(91,"00327D0056127E12")
100 CALL CHAR(93,"FF003412")
```

```

100 A$="4444FFE1E1004444"
110 CALL CHAR(112,A$)
100 A$="FFFFFFFFFFFFFFFF"
110 FOR L=91 TO 96
120 CALL CHAR(L,A$)
130 NEXT L

```

You're now ready to take all of this information and actually design a new character. The 8 x 8 character grid below shows how the *delta* symbol might be defined:

| | Binary | Hex |
|---|-------------|-------|
|  | = 0000 0000 | = 0 0 |
|  | = 0000 0000 | = 0 0 |
|  | = 0000 0000 | = 0 0 |
|  | = 0001 1000 | = 1 8 |
|  | = 0010 0100 | = 2 4 |
|  | = 0100 0010 | = 4 2 |
|  | = 1000 0001 | = 8 1 |
|  | = 1111 1111 | = F F |

The character pattern for this symbol is 00000018244281FF. The pattern has to be assigned to an ASCII character code, so refer to Table 2-1. You can see that the codes 48–57 represent numbers, while codes 65–90 represent uppercase letters. Since you would most likely want to have all of those available, you would pick a code other than one in those two ranges. The [(left bracket) is rarely used; consequently, character code 91 would be a good choice.

The following program displays the new character on the screen:

```

100 CALL CLEAR
110 CALL CHAR(91,"00000018244281FF")
120 DISPLAY AT (10,14):CHR$(91)
130 GOTO 130

```

Line 100 clears the screen. Line 110 defines the new character pattern and assigns it to ASCII code 91. Line 120 DISPLAYs the new character on the screen. The CHR\$ command is used to display the pattern identified with ASCII code 91, which in this case is the delta symbol.

Note that even if you replaced CHR\$(91) in line 120 with [, the delta symbol would still appear. There is a simple reason for this. When you press a key on the console, the computer

1. _____
 2. _____
 3. _____
 4. _____
 5. _____

You can see from the diagram that the four patterns are:

20

```

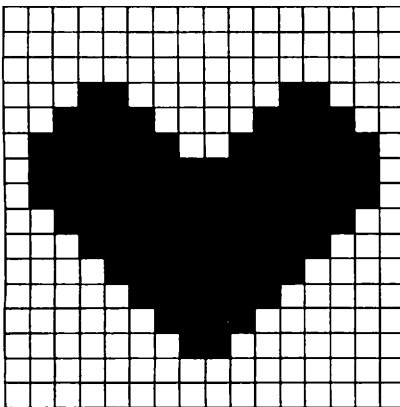
170 NEXT L
180 GOTO 180
190 DATA "03030301010F3F2F"
200 DATA "C0C0C08080F0FCF4"
210 DATA "2723070606060606"
220 DATA "E4C4E06060606060"

```

Line 100 clears the screen. Lines 110–130 read the four DATA statements, which contain the patterns, and assign the patterns to ASCII codes 91 through 94. Line 140 sets up a loop which will be executed four times. Line 150 displays the top half of the robot by putting ASCII characters 91 and 92 side by side at row 10 and the column indicated by L. Line 160 displays the bottom half of the robot in the same fashion using ASCII codes 93 and 94. Line 180 is an infinite loop to keep the robots on the screen.

This same method works for any shape defined with four characters. Figure 2-3 illustrates two common shapes. Try replacing the four DATA statements in the above program with any of the patterns indicated in the figure.

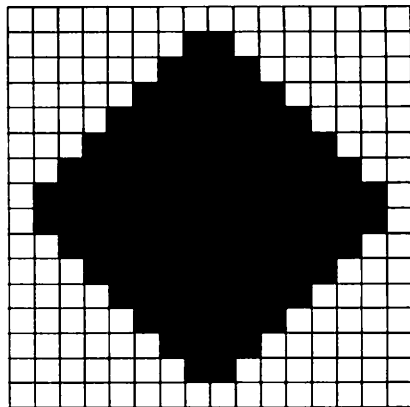
Figure 2-3. Four-Character Shapes



```

DATA "000000183C7E7F7F"
DATA "000000183C7EFEFE"
DATA "3F1F0F0703010000"
DATA "FCF8F0E0C0800000"

```



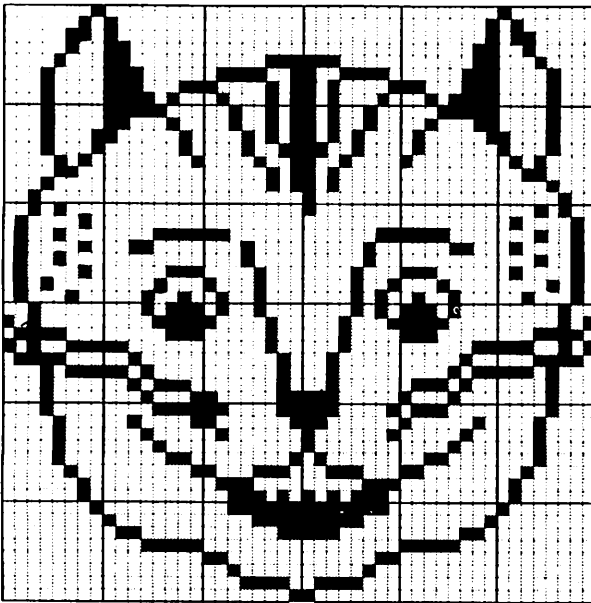
```

DATA "000103070F1F3F7F"
DATA "0080C0E0F0F8FCFE"
DATA "7F3F1F0F07030100"
DATA "FEFCF8F0E0C08000"

```


This method can also be used for shapes which are larger than four characters. The only difference is that you have to define more characters. Program 2-1, "COMPUTE! Cat," produces the display shown in Figure 2-4. Notice that it takes 36 characters to define the friendly feline.

Figure 2-4. COMPUTE! Cat



How COMPUTE! Cat Works

Line(s)

- | | |
|---------|---|
| 100 | Clear the screen. |
| 110-130 | Read the 36 DATA statements and assign them to ASCII character codes 91 through 126. |
| 140-150 | Set up two loops to display six rows and six columns. |
| 160 | Display the ASCII codes 91 through 126 one at a time. When R=1, it displays codes 91-96; when R=2, it displays codes 97-102, etc. |
| 170-180 | Close the loops. |
| 190 | Display COMPUTE! CAT literal. |
| 200 | Infinite loop. |
| 210-560 | DATA statements containing character patterns. |

Program 2-1. COMPUTE! Cat

```

100 CALL CLEAR
110 FOR L=91 TO 126
120 READ A$: CALL CHAR(L,A$)
130 NEXT L
140 FOR R=1 TO 6
150 FOR C=1 TO 6
160 DISPLAY AT(R+9,C+10):CHR$(84+(R*6)+C);
170 NEXT C
180 NEXT R
190 DISPLAY AT(18,7):"C O M P U T E !" :: DISPLAY
    AT(20,12):"C A T"
200 GOTO 200
210 DATA "0102060408101010"
220 DATA "008080C0C0E0E3F4"
230 DATA "0000000007798545"
240 DATA "00000000E09EA1A2"
250 DATA "000101030307C72F"
260 DATA "8040602010080808"
270 DATA "101011110A0C1020"
280 DATA "F8C4820201000000"
290 DATA "2525151309050500"
300 DATA "A4A4A8C890A0A080"
310 DATA "1F23414080000000"
320 DATA "0808888850300804"
330 DATA "2842484248424844"
340 DATA "00000F3000070812"
350 DATA "0000C01010088848"
360 DATA "800003040408090A"
370 DATA "0000F00C00E01048"
380 DATA "1442124212421242"
390 DATA "20A078A778171010"
400 DATA "170F07C020D02E11"
410 DATA "4884040402020283"
420 DATA "0A111010202020E1"
430 DATA "E8F0E003040B7488"
440 DATA "04051EE51EE80808"
450 DATA "1008080808040202"
460 DATA "0E21100806010000"
470 DATA "43814000018E702A"
480 DATA "E2C18280403807AA"
490 DATA "7082040830C00000"
500 DATA "4810101010204040"
510 DATA "0100000000000000"
520 DATA "0080601F00000000"
530 DATA "3F0F0300C0201E01"
540 DATA "FEFCE00003047880"
550 DATA "000106F800000000"
560 DATA "8000000000000000"

```

Using DISPLAY AT

You should have noticed that several of the preceding example programs used the DISPLAY AT command to place characters on the screen. DISPLAY AT is much more versatile than either PRINT or DISPLAY. It allows you to format a screen by placing characters (standard ASCII or graphics) anywhere on the 32 x 24 character grid. In addition, it doesn't produce the normal scrolling action associated with the other commands. DISPLAY AT has the following format:

DISPLAY AT(row,column)[BEEP][ERASE ALL][SIZE(numeric expression)]:variable list

The data identified by variable list is displayed on the 32 x 24 character grid at the location indicated by row and column. Items identified in variable list may be numeric variables, string variables, numeric or string literals, and ASCII character-code patterns returned by the CHR\$ function. For example:

```
100 DISPLAY AT(10,10):A1;" EQUALS ";CHR$(45)
```

BEEP causes a short tone to sound when the data is displayed. ERASE ALL clears the screen before the data is displayed. SIZE is used to blank the number of characters indicated by numeric expression.

Changing Patterns

The TI allows you to change patterns *dynamically* in a program. In other words, the same ASCII character code can be used for more than one pattern definition within the same program. In fact, it can be used as many times as needed. The following short program illustrates this concept:

```
100 CALL CLEAR
110 DISPLAY AT(12,14):CHR$(91)
120 FOR L=1 TO 4
130 FOR L2=1 TO 50 :: NEXT L2
140 READ A$ :: CALL CHAR(91,A$)
150 NEXT L
160 RESTORE :: GOTO 120
170 DATA "1010101010101010"
180 DATA "006030180C060300"
190 DATA "000000FFFF000000"
200 DATA "0003060C18306000"
```

The program displays the standard character pattern (open bracket) for ASCII code 91. Line 120 causes the program to loop four times. Within the loop it reads the alternate patterns (the DATA statements) and changes the pattern for ASCII code 91. Notice that it is not necessary to reDISPLAY the character; it only has to be changed. Line 160 restores the DATA statements at the end of the loop, then reinitializes the loop. The overall effect is to give this single ASCII character code an impression of motion.

Color

Character patterns displayed on the screen may contain two colors. The patterns themselves may be either characters from the standard ASCII character set defined by the TI or characters you have defined yourself. The colors may be any two of the 16 available (Table 2-2). The command that is used to assign these colors is CALL COLOR. It has the following format:

CALL COLOR(character set,foreground color,background color)

Colors are not assigned to individual ASCII character codes. Instead, they are assigned to an ASCII character code set (Table 2-3). Each set, except set 0, consists of eight ASCII character codes. *Character set* is the set number that identifies the eight ASCII characters which will have the assigned color combination. As you can see, there is a limited number of sets. Consequently, it is important to plan ahead when defining your own characters. You should, for example, use ASCII character codes that fall within the same set when defining characters that will have the same color.

Table 2-2. Color Codes

| Color | Code | Color | Code |
|--------------|------|--------------|------|
| Transparent | 1 | Medium Red | 9 |
| Black | 2 | Light Red | 10 |
| Medium Green | 3 | Dark Yellow | 11 |
| Light Green | 4 | Light Yellow | 12 |
| Dark Blue | 5 | Dark Green | 13 |
| Light Blue | 6 | Magenta | 14 |
| Dark Red | 7 | Gray | 15 |
| Cyan | 8 | White | 16 |

Table 2-3. Character Sets

| Set | ASCII codes | Set | ASCII codes |
|-----|-------------|-----|-------------|
| 0 | 30-31 | 8 | 88-95 |
| 1 | 32-39 | 9 | 96-103 |
| 2 | 40-47 | 10 | 104-111 |
| 3 | 48-55 | 11 | 112-119 |
| 4 | 56-63 | 12 | 120-127 |
| 5 | 64-71 | 13 | 128-135 |
| 6 | 72-79 | 14 | 136-143 |
| 7 | 80-87 | | |

Foreground color is the color assigned to the pixels which are *on*. *Background color* is the color assigned to the *off* pixels. In many cases, you would want the background color to be transparent (#1). You can define colors for more than one character set in the same CALL COLOR command. For example,

```
CALL COLOR(8,2,1,9,7,1,10,6,16)
```

assigns a black foreground with a transparent background to set 8 characters, a dark red foreground with a transparent background to set 9 characters, and a light blue foreground with a white background to set 10 characters.

The CALL COLOR command is also used to assign color to sprites. The syntax to accomplish this is similar to that used above and is discussed in detail in the next chapter.

The color of the screen itself can be changed by using the CALL SCREEN command. It has this format:

```
CALL SCREEN(screen color)
```

Screen color is a number from 1 to 16 which represents one of the colors in Table 2-2. The command sets all blank positions on the screen (those that do not contain characters) to the color indicated. The default screen color is cyan (8).

As you use these two commands, you will discover that some color combinations work quite well, but others do not. Black on cyan, for example, provides excellent visible contrast; blue on red, on the other hand, appears smeared and murky. The best way to determine good combinations is to experiment.

Earlier in this chapter you saw how character patterns could be changed dynamically within a program. Character

and screen colors can also be changed this way. The following short program illustrates this capability, as well as the color combinations available:

```
100 CALL CLEAR
110 DISPLAY AT(10,12):"COMPUTE!"
120 FOR L=1 TO 16
130 CALL SCREEN(L)
140 FOR L2=1 TO 16
150 CALL COLOR(5,L2,1,6,L2,1,7,L2,1)
160 FOR D=1 TO 400 :: NEXT D
170 NEXT L2 :: NEXT L
```

Bars and Lines

Thus far, discussion has centered on defining and displaying each individual character on the screen. Often, however, you will want to use a single character many times—for instance, when you are drawing bars, lines, or diagonals.

Your TI has two commands which were specifically designed for this type of programming. They are CALL HCHAR and CALL VCHAR. CALL HCHAR lets you repeat a single character horizontally across the screen; CALL VCHAR lets you repeat a single character vertically. They have the following format:

```
CALL HCHAR(row,column,character-code,repetition) CALL
VCHAR(row,column,character-code,repetition)
```

The character's initial location on the 32 x 24 screen grid is defined by *row* and *column*. *Row* has a value from 1 to 24; *column* has a value from 1 to 32. *Character code* is the ASCII character code used to define the desired pattern. *Repetition* indicates the number of characters that will be placed on the screen horizontally or vertically. The four parameters may be numeric literals or numeric variables. The following examples illustrate how these commands may be used:

```
100 CALL CLEAR
110 CALL CHAR(96,"FFFFFFFFFFFFFFFF")
120 CALL HCHAR(3,1,96,10)
130 GOTO 130

100 CALL CLEAR
110 CALL CHAR(96,"FFFFFFFFFFFFFFFF")
120 CALL CHAR(97,"0000FFFFFFFF0000")
```


Introduction to Graphics

```
130 CALL HCHAR(10,4,96,18)
140 CALL VCHAR(2,19,97,18)
150 GOTO 150

100 CALL CLEAR
110 CALL CHAR(96,"000000FFFF000000")
120 FOR L=1 TO 4
130 CALL HCHAR(L*2,L+6,96,L*3)
140 NEXT L
150 GOTO 150
```

These commands are very useful programming tools. Depending on the character pattern, they can be used to draw solid bars, lines, chains, rectangles, screen borders—in fact, anything that requires repetitive characters. In addition, they allow you to use all 32 columns on the screen. PRINT and DISPLAY let you use only 28 columns (columns 3 through 30).

The following example, "Histogram," demonstrates how these two instructions can be used to write a program. The program produces a bar graph showing computer sales over a 12-month period. You may enter values yourself or request the demonstration mode which generates values randomly. If you enter the values, the program will automatically pick the correct scale. The maximum value you may enter is 10000.

How Histogram Works

Line(s)

| | |
|---------|---|
| 100 | Clear the screen. |
| 110 | DIM statement for the 12 values allowed on the bar chart. |
| 120–190 | Define the character patterns used in the program. ASCII codes 96–99 define the characters used for the actual bars. Code 96 is a complete block; 97 is a 3/4 block; 98 is a 1/2 block; and 99 is a 1/4 block. Using four blocks in quarter steps allows higher resolution on the bars. ASCII code 43 defines the Y-axis and tick marks. Codes 104 and 112 define the horizontal marking lines. |
| 200 | Assign colors to the various characters. |
| 210–250 | Display a selection menu. |
| 260–280 | Draw X and Y axes and display literals. |
| 290–300 | Draw the marking lines in two colors. |

- 310-320 Display X and Y axes identification.
- 330 Determine the maximum value to be graphed. This is used to determine the scaling.
- 340-360 Determine scale.
- 370-450 Draw the bars for the values on the graph. Lines 410-440 determine which 1/4 block size is to be used on the top of each bar to make it more accurate.
- 460 Display message indicating the scale used.
- 470-500 End of program control.
- 510-590 Accept values from keyboard.
- 600-630 Generate random values for demonstration.

Program 2-2. Histogram

```

100 CALL CLEAR
110 DIM V(12)
120 CALL CHAR(96,"FFFFFFFFFFFFFFFF")
130 CALL CHAR(97,"0000FFFFFFFFFFFF")
140 CALL CHAR(98,"00000000FFFFFFFF")
150 CALL CHAR(99,"000000000000FFFF")
160 CALL CHAR(43,"FFF0F0F0F0F0F0F0")
170 CALL CHAR(104,"FF00000000000000")
180 CALL CHAR(112,"FF00000000000000")
190 CALL CHAR(42,"FFFFFFFFFFFFFFFF")
200 CALL COLOR(9,5,1,10,3,1,11,7,1)
210 DISPLAY AT(4,1):"1 - ENTER VALUES MANUALLY" ::
    DISPLAY AT(6,1):"2 - RUN DEMONSTRATION"
220 DISPLAY AT(8,1):"WHICH ONE? --> "
230 ACCEPT AT(8,15)VALIDATE("12")SIZE(-1)BEEP:OPT
240 ON OPT GOSUB 510,600
250 CALL CLEAR
260 DISPLAY AT(2,10):"COMPUTER SALES" :: DISPLAY AT
    T(4,13):"BY MONTH"
270 CALL HCHAR(18,6,42,25)
280 CALL VCHAR(8,6,43,10)
290 FOR L=8 TO 12 :: CALL HCHAR(L,7,104,24):: NEXT
    L
300 FOR L=13 TO 17 :: CALL HCHAR(L,7,112,24):: NEX
    T L
310 DISPLAY AT(19,6):"J F M A M J J A S O N D"
320 DISPLAY AT(8,1):"10":: DISPLAY AT(13,2):"5"::
    : DISPLAY AT(17,2):"1"
330 MX=0 :: FOR L=1 TO NV :: MX=MAX(V(L),MX):: NEX
    T L

```

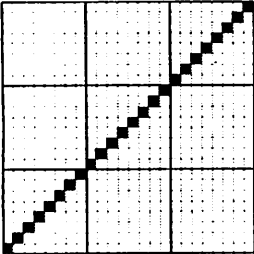
Introduction to Graphics

```
340 IF MX<=10 THEN S=1
350 IF MX>10 AND MX<=1000 THEN S=100 :: LIT$="HUND
    REDS"
360 IF MX>1000 THEN S=1000 :: LIT$="THOUSANDS"
370 FOR L=1 TO NV
380 BAR=INT(V(L)/S):: FR=V(L)/S-INT(V(L)/S)
390 CALL VCHAR(18-BAR,6+(L*2),96,BAR)
400 FRP=18-BAR-1
410 IF FR>=.15 AND FR<.35 THEN CALL VCHAR(FRP,6+(L
    *2),99,1)
420 IF FR>=.35 AND FR<.65 THEN CALL VCHAR(FRP,6+(L
    *2),98,1)
430 IF FR>=.65 AND FR<.9 THEN CALL VCHAR(FRP,6+(L*
    2),97,1)
440 IF FR>=.9 THEN CALL VCHAR(FRP,6+(L*2),96,1)
450 NEXT L
460 IF S>1 THEN DISPLAY AT(21,7):"IN ";LIT$;" OF U
    NITS"
470 DISPLAY AT(24,9):"PRESS ANY KEY"
480 CALL KEY(3,K,S):: IF S=0 THEN 480
490 CALL CLEAR
500 GOTO 210
510 CALL CLEAR
520 DISPLAY AT(4,1):"NUMBER OF VALUES 1-12? -->"
530 ACCEPT AT(4,27)VALIDATE(NUMERIC)BEEP:NV
540 IF NV>12 THEN 530
550 FOR L=1 TO NV
560 DISPLAY AT(6+L,1):"VALUE # ";L
570 ACCEPT AT(6+L,13)VALIDATE(NUMERIC)BEEP:V(L)::
    IF V(L)>10000 THEN 570
580 NEXT L
590 RETURN
600 FOR L=1 TO 12
610 V(L)=INT(1+RND*10)
620 NEXT L
630 NV=12 :: RETURN
```

Diagonals

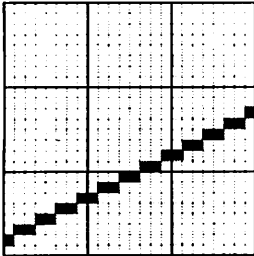
In many cases the bar or line you want to draw is not horizontal or vertical, but diagonal. One way to draw a diagonal is to define each character in the diagonal's path. Sometimes only one character definition is required, as illustrated by the first diagonal in Figure 2-5. This character, once defined, is repeated for the length of the diagonal. In other cases, more than one character definition is required to draw the diagonal, and a *sequence* of characters is repeated along the length of the diagonal.

Figure 2-5. Drawing Diagonals



45-degree angle
Requires one character definition:

"0102040810204080"



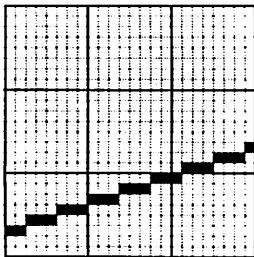
30-degree angle (approximately)

Requires three character definitions:

"0000010618608000"

"1860800000000000"

"0000000000000106"



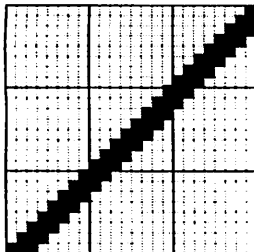
20-degree angle (approximately)

Requires three character definitions:

"0000000738C00000"

"031CE00000000000"

"00000000000010E70"



45-degree angle with thick line

Requires two character definitions:

"070E1C3870E0C080"

"0000000000000103"

Introduction to Graphics

This method gives you a lot of control over the size and resolution of the diagonal. By defining the characters and placing them on the screen, you can create diagonal lines, zigzag, geometric figures, and the like.

Sometimes, it is possible to use CALL HCHAR, CALL VHCHAR, or even DISPLAY AT to draw diagonals. It depends primarily on the angle and the degree of resolution desired. The following program shows how this might be done.

```
100 CALL CLEAR
110 CALL CHAR(42,"FFFFFFFFFFFFFFFF")
120 CALL CHAR(96,"FFFFFFFFFFFFFFFF")
130 CALL HCHAR(19,2,96,25)
140 CALL VCHAR(2,2,96,25)
150 CALL COLOR(2,5,1)
160 FOR X=1 TO 18
170 C=X
180 Y=X
190 IF Y>18 OR Y=0 THEN 220
200 R=18-Y+1
210 DISPLAY AT(R,C):CHR$(42)
220 NEXT X
230 GOTO 230
```

The program simply draws an X and Y axis and plots a line based on the equation in line 180. As you can see when you RUN the program, the line produced is a 45-degree angle with *low resolution*. If you change the pattern for ASCII code 42 to 0102040810204080, the program will produce a *high-resolution*, 45-degree angle line.

But what if you wanted to change the equation in line 180 to $Y=2*X$? Several things occur as a result of this change. First, the high-resolution pattern for ASCII code 42 no longer works. It worked originally only because the angle of the line was known and the pattern fit that angle. The line represented by the new equation, however, has a different angle which is unknown. Consequently, the pattern for ASCII code 42 must be changed back to all F's.

The second thing affected by the change is the continuity of the line: The line has the correct angle, but it is not continuous. This happens because the program draws the line by plotting individual points, and the new equation results in a greater distance between these plotted points. Increasing the

number of iterations in the program loop by adding STEP .5 to line 160 increases the number of points plotted and, consequently, decreases the distance between them. The resulting line is continuous, but its resolution is quite low.

Character Concatenation

There are many cases when consecutive screen columns do not contain the same character. For example, assume you had five consecutive characters starting at column 1 on row 5 and that each character required a different bit pattern. How would you handle such a situation?

There are several possible approaches. You could use five CALL HCHAR or DISPLAY AT statements to display the characters one at a time, or you could display the characters by using the CALL HCHAR or DISPLAY AT in a loop and incrementing the column position.

But there is one other method that is faster and requires fewer program lines. The characters can be *concatenated* into a single character string. The TI lets you do this with the ampersand symbol. Once the characters are concatenated, you can use a single DISPLAY AT to display the entire string on the screen.

The following program, "Plane," illustrates how the speed inherent in this method can be used to simulate motion. A four-character airplane, ten-character rope, and eight-character sign are concatenated. Forty spaces are concatenated to each end of this character-string. By DISPLAYing the string 28 characters at a time within a loop, and by shifting the starting point of the string one place to the right for each iteration, the plane appears to fly across the top of the screen.

How Plane Works

Line(s)

- | | |
|---------|--|
| 100 | Clear the screen. |
| 110-250 | Define the characters used in the program. Character codes 94-97 define the plane; 98 defines one length of rope; 40 defines the ground; and 128-130 and 136-138 define the house on the ground. |
| 260 | Assign the colors used by the various characters. |
| 270-330 | Build the seven character strings which make up the house. |

Introduction to Graphics

- 340-370 Build the character string which consists of the plane, rope, sign, and blank spaces. The four characters of the plane are added to ten repetitions of the rope character, and the word COMPUTE! is added to this. Forty spaces are added to each end to allow the plane to travel across the screen.
- 380 Display the ground on the screen.
- 390-410 Display on the screen the strings which make up the house.
- 420-450 Move the plane, rope, and sign across the screen. This is accomplished by repeatedly displaying 28 characters of the string within the loop. The start location in the string is moved one position to the right for each iteration of the loop, thus creating the motion.
- 460 Repeat the process.

Program 2-3. Plane

```
100 CALL CLEAR
110 CALL CHAR(94,"40405F7F7F4F4000")
120 CALL CHAR(95,"0002FBFFFFFF2070")
130 CALL CHAR(96,"0000FFFFFFFF0000")
140 CALL CHAR(97,"1E3EFEFEFCC04020")
150 CALL CHAR(98,"000000FF00000000")
160 CALL CHAR(40,"FFFFFFFFFFFFFFFF")
170 CALL CHAR(104,"FFFFFFE7E7FFFFFF")
180 CALL CHAR(112,"183C7EFFFFFFFF")
190 CALL CHAR(120,"FFBBFFFBFFBB")
200 CALL CHAR(128,"FFFFFFFFFFFFFFFF")
210 CALL CHAR(129,"0103070F1F3F7FFF")
220 CALL CHAR(130,"80C0E0F0F8FCFEFF")
230 CALL CHAR(136,"FF1010FF101010FF")
240 CALL CHAR(137,"FFFFFFFFFFFFFFFD")
250 CALL CHAR(138,"EDFFFFFFFFFFFF"):: CALL CHAR(
64,"0018181818180018")
260 CALL COLOR(2,3,1,5,7,16,6,7,16,7,7,16,10,2,16,
11,2,1,12,2,16,13,5,1,14,15,2)
270 H$(1)=RPT$(" ",2)&CHR$(129)&CHR$(130)
280 H$(2)=" "&CHR$(129)&CHR$(128)&CHR$(128)&CHR$(1
30)
290 H$(3)=CHR$(129)&RPT$(CHR$(128),4)&CHR$(130)
300 H$(4)=CHR$(128)&CHR$(136)&RPT$(CHR$(128),2)&CH
R$(136)&CHR$(128)
310 H$(5)=RPT$(CHR$(128),6)
320 H$(6)=CHR$(128)&CHR$(136)&CHR$(128)&CHR$(137)&
RPT$(CHR$(128),2)
```

```
330 H$(7)=RPT$(CHR$(128),3)&CHR$(138)&RPT$(CHR$(128),2)
340 PL$=RPT$(" ",40)
350 PL$=PL$&CHR$(94)&CHR$(95)&CHR$(96)&CHR$(97)&RPT$(CHR$(98),10)
360 PL$=PL$&"COMPUTE"&CHR$(64)
370 PL$=PL$&RPT$(" ",40)
380 FOR L=22 TO 24 :: CALL HCHAR(L,1,40,32):: NEXT L
390 FOR L=1 TO 7
400 DISPLAY AT(14+L,16):H$(L)
410 NEXT L
420 FOR L=1 TO 120
430 DISPLAY AT(3,1):SEG$(PL$,L,28)
440 FOR D=1 TO 25 :: NEXT D
450 NEXT L
460 GOTO 420
```

Special Effects

You can enhance your programs, particularly your games, by adding special effects to them. Special effects can range from simulated motion and animation to laser beams and explosions. They are used to give programs action and a sense of realism. You have already seen how to make a plane fly across the screen. Now consider some other possibilities.

The program below illustrates how an alien ship can be made to fire its laser, using only those programming techniques that have been discussed so far.

```
100 CALL CLEAR :: CALL SCREEN(2)
110 CALL CHAR(96,"00C0FCFFFFFFCC000")
120 CALL CHAR(112,"000000FFFF000000")
130 CALL COLOR(9,6,1,11,9,1)
140 FOR L=2 TO 14
150 CALL HCHAR(L-1,4,32,1) :: CALL HCHAR(L,4,96,1)
160 NEXT L
170 FOR L=1 TO 4
180 CALL HCHAR(14,5,112,25)
190 CALL HCHAR(14,5,32,25)
200 NEXT L
210 FOR L=1 TO 500 :: NEXT L :: STOP
```

Line 100 clears the screen and sets it to black. Line 110 defines the alien spaceship; line 120 defines the laser beam. Line 130 sets the ship's color to blue and the laser to red. The

Introduction to Graphics

ship is brought down from the top of the screen by lines 140–160. Lines 170–200 fire the laser four times. The program ends on line 210.

Program 2-4, "Blinky," demonstrates that the computer doesn't have to be a faceless collection of circuits and chips. As you will see when you RUN the program, your TI can be quite endearing.

How Blinky Works

Line(s)

| | |
|---------|---|
| 100 | Clear the screen. |
| 110–160 | Define the characters used in the program. ASCII code 96 is the border for the face. A\$ contains the pattern for the open eyes; B\$ contains the pattern for the closed eyes. Codes 97 and 98 define the corners of the smile. |
| 170–210 | Draw the face on the screen. |
| 220–270 | Cause the eyes to blink five times. |
| 280–300 | Change the corners of the mouth to make a smile. |
| 310–340 | Cause the face to wink at the observer. |
| 350–370 | End the program. |

Program 2-4. Blinky

```
100 CALL CLEAR
110 CALL CHAR(96,"FFFFFFFFFFFFFFFF")
120 A$="FF818181818181FF" :: B$="000000FFFF000000"
130 CALL CHAR(97,"0303030300000000")
140 CALL CHAR(98,"C0C0C0C000000000")
150 CALL CHAR(104,A$)
160 CALL CHAR(112,B$)
170 CALL HCHAR(10,12,96,8):: CALL HCHAR(18,12,96,8)
180 CALL VCHAR(11,12,96,7):: CALL VCHAR(11,19,96,7)
190 CALL HCHAR(12,14,104,1):: CALL HCHAR(12,17,104,1)
200 CALL HCHAR(14,15,97,1):: CALL HCHAR(14,16,98,1)
210 CALL HCHAR(16,14,112,4)
220 FOR L=1 TO 5
230 FOR L2=1 TO 600 :: NEXT L2
```

```
240 CALL CHAR(104,B$)
250 FOR L2=1 TO 40 :: NEXT L2
260 CALL CHAR(104,A$)
270 NEXT L
280 FOR L=1 TO 100 :: NEXT L
290 CALL HCHAR(16,14,97,1):: CALL HCHAR(16,17,98,1
)
300 FOR L2=1 TO 700 :: NEXT L2
310 CALL HCHAR(12,14,112,1)
320 FOR L2=1 TO 80 :: NEXT L2
330 CALL HCHAR(12,14,104,1)
340 FOR L=1 TO 200 :: NEXT L

350 DISPLAY AT(20,10): "BYE NOW!"
360 FOR L=1 TO 500 :: NEXT L
370 CALL CLEAR :: STOP
```

The last program in this section, "Tank Attack," illustrates how you can simulate a battle scene on the TI. In this program, an artillery piece is firing in the direction of an opposing tank. The tank returns the fire, and the artillery piece is destroyed. Figure 2-6 shows the character patterns used in this simulation.

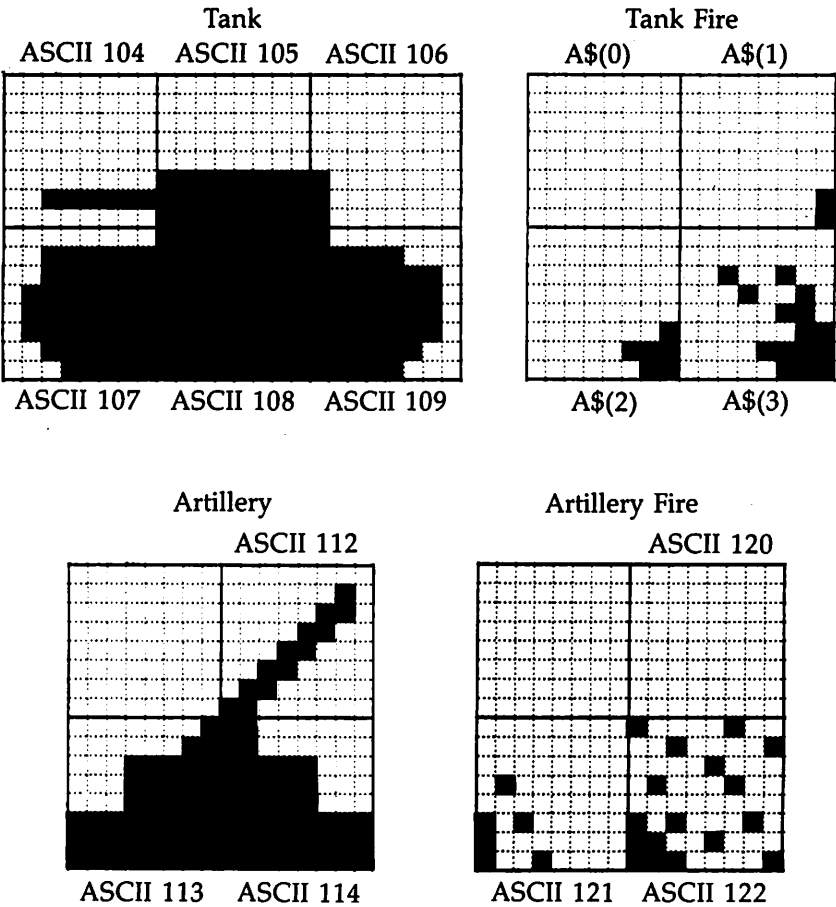
How Tank Attack Works

| Line(s) | |
|---------|--|
| 100 | Clear the screen. |
| 110-240 | Define the characters used in the program. ASCII code 42 is the base of the display. The tank is defined by the six ASCII codes 104-109. The artillery piece is defined by 112-114. ASCII codes 120-122 are used to simulate the artillery piece firing. A\$(0)-A\$(3) contain the character patterns to simulate the tank firing. |
| 250-260 | Assign the colors used for the character. |
| 270 | Draw the base of the screen. |
| 280-290 | Draw the tank. |
| 300 | Draw the artillery piece. |
| 310 | Timing loop. |
| 320-370 | Cause the artillery piece to fire four times. The loop changes the pattern in front of the gun's muzzle rapidly, giving the appearance of firing. |
| 380-450 | Cause the tank to return fire four times. Again, the loop causes the pattern to change rapidly, simulating return fire. |

460-490 This loop alternately displays codes 121 and 122 at the positions where the artillery piece was located. This causes the artillery piece to explode.

500-510 End of program.

Figure 2-6. Tank Attack Character Patterns



Program 2-5. Tank Attack

```

100 CALL CLEAR
110 CALL CHAR(42,"FFFFFFFFFFFFFFFF")
120 CALL CHAR(104,"00000000000003F00")
130 CALL CHAR(105,"0000000000FFFFFFFF")
140 CALL CHAR(106,"00000000000808080")
150 CALL CHAR(107,"001F1F7F7F7F3F1F")
160 CALL CHAR(108,"FFFFFFFFFFFFFFFF")
170 CALL CHAR(109,"80F8F8FEFEFEFCF8")
180 A$(0)="0000000000000000" :: A$(1)="000000000000
00101" :: A$(2)="00000000000010703" :: A$(3)="0
00000241204030F07"
190 CALL CHAR(112,"0002060C183060C0")
200 CALL CHAR(113,"01031F1F1FFFFFFFF")
210 CALL CHAR(114,"C0E0F0F0F0FFFFFFFF")
220 CALL CHAR(121,"0000004000A08090")
230 CALL CHAR(122,"8421084400A2C8E1")
240 CALL CHAR(120,A$(0))
250 CALL COLOR(2,13,1,12,9,1)
260 CALL SCREEN(6)
270 FOR L=22 TO 24 :: CALL HCHAR(L,1,42,32):: NEXT
L
280 FOR L=104 TO 106 :: DISPLAY AT(20,L-80):CHR$(L
);:: NEXT L
290 FOR L=107 TO 109 :: DISPLAY AT(21,L-83):CHR$(L
);:: NEXT L
300 DISPLAY AT(20,4):CHR$(112);:: DISPLAY AT(21,3)
:CHR$(113);:: DISPLAY AT(21,4):CHR$(114);
310 FOR L=1 TO 1000 :: NEXT L
320 FOR L=1 TO 4
330 DISPLAY AT(19,5):CHR$(121);
340 DISPLAY AT(19,5):CHR$(122);
350 DISPLAY AT(19,5):" ";
360 FOR L2=1 TO 400 :: NEXT L2
370 NEXT L
380 DISPLAY AT(20,23):CHR$(120);
390 FOR X=1 TO 4
400 FOR L2=1 TO 300 :: NEXT L2
410 FOR L=1 TO 3
420 CALL CHAR(120,A$(L))
430 NEXT L
440 CALL CHAR(120,A$(0))
450 NEXT X
460 FOR L=1 TO 7
470 DISPLAY AT(20,4):CHR$(121);:: DISPLAY AT(21,4)
:CHR$(121);:: DISPLAY AT(21,3):CHR$(121);
480 DISPLAY AT(20,4):CHR$(122);:: DISPLAY AT(21,4)
:CHR$(122);:: DISPLAY AT(21,3):CHR$(122);

```

```
490 NEXT L
500 DISPLAY AT(20,4):" ";; DISPLAY AT(21,3):" ";
510 GOTO 510
```

High-Resolution Graphics

It is enjoyable, even exciting, to watch high-resolution graphics. A highly detailed graphic display will attract attention faster than any other type of demonstration. Sometimes you can even hear the *oohs* and *aahs*.

One of the main reasons for the popularity of the PARSEC command module, for example, is its detailed multicolor graphics. Achieving such detail and resolution usually requires the use of Assembler Language. But this doesn't mean that you need the Editor/Assembler package to do hi-res graphics on your TI. You can actually do quite a bit from BASIC.

The biggest drawback to using BASIC is that you can't access each of the 49,152 pixels directly. You can, however, access each bit in a single 8 x 8 character. By defining and putting enough of these characters together, you can produce some very nice hi-res images on your TV or monitor.

Figure 2-7. 3-D Shapes

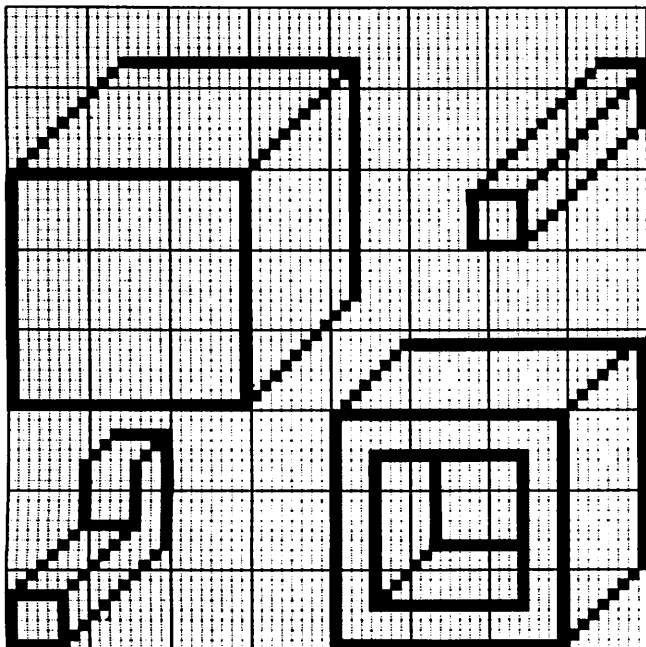


Figure 2-7 shows some three-dimensional shapes drawn on graph paper. The area shown is eight characters by eight characters. Each character is made up of an 8 x 8 matrix. By assigning each character to a unique ASCII character code and then defining the pattern for each character, you can write a program to display these 3-D shapes on the screen. Program 2-6 illustrates how it can be done.

Program 2-6. 3-D Shapes

```

100 CALL CLEAR
110 FOR L=33 TO 44
120 READ A$ :: CALL CHAR(L,A$)
130 NEXT L
140 FOR L=91 TO 142
150 READ A$ :: CALL CHAR(L,A$)
160 NEXT L
170 FOR R=1 TO 8
180 FOR C=1 TO 8
190 CHAR=(R-1)*8+C+32
200 IF CHAR>44 THEN CHAR=CHAR+46
210 DISPLAY AT(R+8,C+8):CHR$(CHAR);
220 NEXT C :: NEXT R :: DISPLAY AT(20,9):"3-D SHAP
    ES"
230 GOTO 230
240 DATA "000000000000000000"
250 DATA "000000000001F2040"
260 DATA "00000000000FF0000"
270 DATA "00000000000FF0000"
280 DATA "00000000000E060A0"
290 DATA "00000000000000000"
300 DATA "00000000000000000"
310 DATA "000000000001F2143"
320 DATA "0001020408102040"
330 DATA "80000000000000000"
340 DATA "00000000000000000"
350 DATA "0102040810204080"
360 DATA "2020202020202020"
370 DATA "00000000000000000"
380 DATA "0001020408102142"
390 DATA "8509112142840810"
400 DATA "FF80808080808080"
410 DATA "FF00000000000000"
420 DATA "FF01010101010101"
430 DATA "00000000000000000"
440 DATA "2020202020202020"
450 DATA "0001030202020203"
460 DATA "8408F011121418F0"

```


Introduction to Graphics

```
470 DATA "2040800000000000"
480 DATA "8080808080808080"
490 DATA "0000000000000000"
500 DATA "0101010101010101"
510 DATA "0000000000000001"
520 DATA "2020202020408000"
530 DATA "0000000000000000"
540 DATA "0000000000000000"
550 DATA "0000000000000000"
560 DATA "80808080808080FF"
570 DATA "00000000000000FF"
580 DATA "01010101010101FF"
590 DATA "0204081020408000"
600 DATA "0001020408102040"
610 DATA "00FF000000000000"
620 DATA "00FF000000000000"
630 DATA "00FF030509112141"
640 DATA "00000000000010101"
650 DATA "00003F4385F90909"
660 DATA "0000000000000000"
670 DATA "0000000000000000"
680 DATA "FF8080808F888888"
690 DATA "FF000000FF202020"
700 DATA "FF010101F1111111"
710 DATA "8101010101010101"
720 DATA "0101010102040810"
730 DATA "090909F911214284"
740 DATA "0000000000000000"
750 DATA "0000000000000000"
760 DATA "8888888888888888"
770 DATA "20202020203F4080"
780 DATA "1111111111F11111"
790 DATA "0101010101010101"
800 DATA "2142FC84848586FC"
810 DATA "0810204080000000"
820 DATA "0000000000000000"
830 DATA "0000000000000000"
840 DATA "898A8C8F808080FF"
850 DATA "000000FF000000FF"
860 DATA "111111F1010101FF"
870 DATA "0204081020408000"
```

The second method for producing hi-res graphics on the TI doesn't require you to predefine the image with CALL CHAR statements. Instead, the TI itself will draw the image based on an equation. It will determine which bits are needed, and then it will turn them on.

This method still starts with the 64 ASCII character codes used in the previous program. Rather than defining these characters, however, all bits are initially turned off (set to zero). These characters are located on the screen in the 8 x 8 arrangement shown in Figure 2-8. Of course, since all the bits are turned off, nothing shows up on the screen at this point.

Figure 2-8. Character Arrangement for Graph

| | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|
| 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 41 | 42 | 43 | 44 | 91 | 92 | 93 | 94 |
| 95 | 96 | 97 | 98 | 99 | 100 | 101 | 102 |
| 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 |
| 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 |
| 119 | 120 | 121 | 122 | 123 | 124 | 125 | 126 |
| 127 | 128 | 129 | 130 | 131 | 132 | 133 | 134 |
| 135 | 136 | 137 | 138 | 139 | 140 | 141 | 142 |

This arrangement is analogous to a 64 x 64 pixel matrix on the screen. The equation determines which pixels in the 64 x 64 matrix must be turned on. But since the pixels can't be accessed directly, the 8 x 8 character in which they reside must be found. The pattern for that character is then changed to reflect the *on* condition of the pixel. Slowly, the image is drawn on the screen.

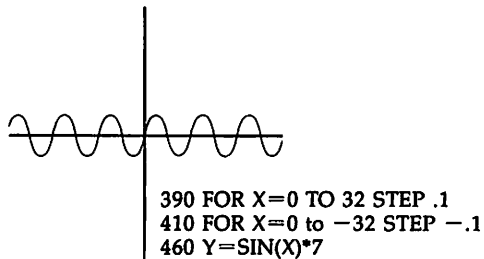
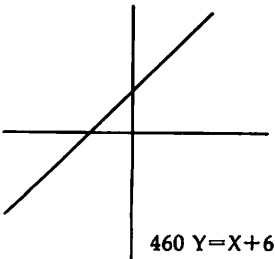
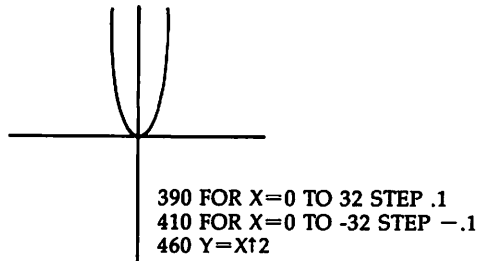
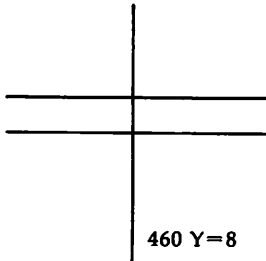
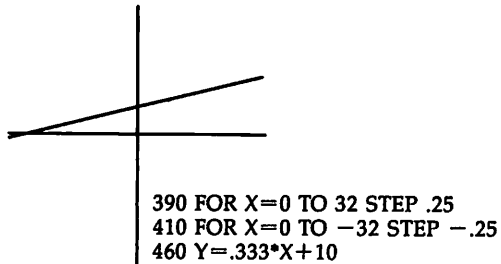
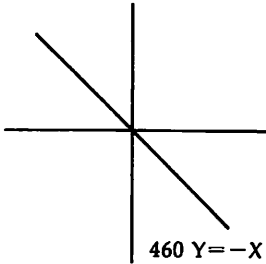
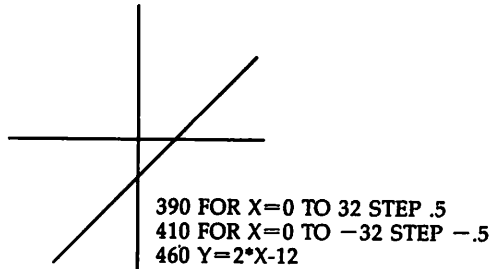
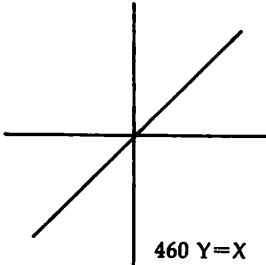
Program 2-7, "Graph," demonstrates how this technique works. It may seem a bit complex at first, but if you study the program, it should start to make sense.

There are two things to keep in mind when using this routine. First, due to the number of computations required to find and set the appropriate pixels, program execution is quite slow. This is especially true with STEPs of .5 or less. The second thing to keep in mind is that the technique uses a lot of memory, due to the large number of character definitions.

Figure 2-9 shows what you'll get if you modify the program for different equations.

Figure 2-9. Program Changes for Graph

Change lines to:



How Graph Works

| Line(s) | |
|---------|---|
| 100 | DIMension arrays. |
| 110-120 | Clear the screen and display message. |
| 130-140 | Set up character patterns. BL\$ is used to blank out the characters used by the program. A\$, B\$, C\$, and D\$ are used to insert the X and Y axes in the display area. |
| 150 | Load four-digit bit strings into BITS\$. |
| 160-170 | Blank out the ASCII characters used in the display area. |
| 180-230 | Insert the X and Y axis lines in the display area. |
| 240 | Display the X and Y axis values. |
| 250-320 | Load the A array with the ASCII character codes in the display area. This is used later to determine which ASCII character code to change. |
| 330 | Erase STAND BY message. |
| 340-380 | Display the 64 characters on the screen 8 characters at a time. This builds the 8 x 8 display area. |
| 390-400 | Initiate the routine to plot the points for the positive values of X. |
| 410-420 | Initiate the routine to plot the points for the negative values of X. |
| 430-450 | End of program. |
| 460 | This is the equation which controls where the points are plotted on the graph. Figure 2-9 shows possible alternatives. |
| 470-490 | Determine the actual location of the point in the 64 x 64 pixel matrix. The program plots both positive and negative values, but negatives in the actual matrix aren't possible, hence the +32. |
| 500-520 | Determine the location of the pixel in the 8 x 8 character matrix. This must be done since the pixel cannot be accessed directly. |
| 530 | Get the hex pattern of the character that contains the pixel. The A array contains the ASCII codes of the characters used. The pattern is placed in W\$. |
| 540-560 | Determine which byte in the pattern contains the pixel. |

Introduction to Graphics

- 570-620 Convert the one-byte hex pattern which contains the pixel into an eight-digit bit pattern (for example, 00100110).
- 630-660 C1 contains the column position of the pixel within the 8 x 8 character grid. If the bit in this position is not on, it is turned on. If it is on, it stays as is.
- 670-690 Convert the bit pattern, with the updated pixel, back into a hex pattern.
- 700-730 Combine the new one-byte hex pattern with the other seven bytes in the pattern.
- 740 Change the pattern of the ASCII character code which effectively plots the point on the screen.
- 750 Return from plot subroutine.
- 760-770 Hex pattern binary equivalents.

Program 2-7. Graph

```
100 DIM A(8,8),BITS$(15)
110 CALL CLEAR
120 DISPLAY AT(4,6):"PLEASE STAND BY"
130 BL$="0000000000000000" :: HEX$="0123456789ABCD
    EF"
140 A$="0101010101010101" :: B$="8080808080808080"
    :: C$="00000000000000FF" :: D$="FF000000000000
    000000"
150 FOR L=0 TO 15 :: READ BITS$(L):: NEXT L
160 FOR L=33 TO 44 :: CALL CHAR(L,BL$):: NEXT L
170 FOR L=91 TO 143 :: CALL CHAR(L,BL$):: NEXT L
180 CALL CHAR(36,A$,44,A$,98,A$,122,A$,130,A$,138,
    A$)
190 CALL CHAR(37,B$,91,B$,99,B$,123,B$,131,B$,139,
    B$)
200 CALL CHAR(103,C$,104,C$,105,C$,106,C$,109,C$,1
    10,C$)
210 CALL CHAR(111,D$,112,D$,113,D$,116,D$,117,D$,1
    18,D$)
220 CALL CHAR(106,"01010101010101FF"):: CALL CHAR(
    107,"80808080808080FF")
230 CALL CHAR(114,"FF010101010101"):: CALL CHAR(
    115,"FF80808080808080")
240 DISPLAY AT(6,13):"32" :: DISPLAY AT(11,19):"32
    " :: DISPLAY AT(11,7):"-32":: DISPLAY AT(17,1
    2):"-32"
250 X=33
260 FOR L=1 TO 8
```

```

270 FOR L2=1 TO 8
280 IF X=45 THEN X=91
290 A(L,L2)=X
300 X=X+1
310 NEXT L2
320 NEXT L
330 DISPLAY AT(4,1):" "
340 FOR L=1 TO 8
350 L$=""
360 FOR X=1 TO 8 :: L$=L$&CHR$(A(L,X)):: NEXT X
370 DISPLAY AT(7+L,10):L$;
380 NEXT L
390 FOR X=0 TO 32 STEP .3
400 GOSUB 460 :: NEXT X
410 FOR X=0 TO -32 STEP -.3
420 GOSUB 460 :: NEXT X
430 DISPLAY AT(22,8):"PRESS ANY KEY"
440 CALL KEY(3,K,S):: IF S=0 THEN 440
450 CALL CLEAR :: STOP
460 Y=X
470 XX=INT(X+32):: YY=INT(Y+32)
480 IF XX>64 OR XX<=0 THEN 750
490 IF YY>64 OR YY<=0 THEN 750
500 R=64-YY+1
510 CC=INT((XX-1)/8)+1
520 RR=INT((R-1)/8)+1
530 CALL CHARPAT(A(RR,CC),W$)
540 C1=XX-((CC-1)*8)
550 R1=INT((64-YY)-((RR-1)*8)+1)
560 BYTE$=SEG$(W$,R1*2-1,2)
570 EXPAND$=""
580 FOR L3=1 TO 2
590 HOLD$=SEG$(BYTE$,L3,1)
600 IF HOLD$>"9" THEN POINT=ASC(HOLD$)-55 ELSE POINT=VAL(HOLD$)
610 EXPAND$=EXPAND$&BITS$(POINT)
620 NEXT L3
630 NEWPAT$=""
640 FOR L2=1 TO 8
650 IF C1<>L2 THEN NEWPAT$=NEWPAT$&SEG$(EXPAND$,L2,1)ELSE NEWPAT$=NEWPAT$&"1"
660 NEXT L2
670 HIGH$=SEG$(HEX$,8*VAL(SEG$(NEWPAT$,1,1))+4*VAL(SEG$(NEWPAT$,2,1))+2*VAL(SEG$(NEWPAT$,3,1))+VAL(SEG$(NEWPAT$,4,1))+1,1)
680 LOW$=SEG$(HEX$,8*VAL(SEG$(NEWPAT$,5,1))+4*VAL(SEG$(NEWPAT$,6,1))+2*VAL(SEG$(NEWPAT$,7,1))+VAL(SEG$(NEWPAT$,8,1))+1,1)
690 BYTE$=HIGH$&LOW$

```

```
700 NEWCHAR$=""
710 FOR L2=1 TO 16
720 IF L2<>R1*2-1 THEN NEWCHAR$=NEWCHAR$&SEG$(W$,L
    2,1)ELSE NEWCHAR$=NEWCHAR$&BYTE$ :: L2=L2+1
730 NEXT L2
740 CALL CHAR(A(RR,CC),NEWCHAR$)
750 RETURN
760 DATA "0000","0001","0010","0011","0100","0101"
    ,"0110","0111","1000"
770 DATA "1001","1010","1011","1100","1101","1110"
    ,"1111"
```

3

Sprites



Chapter 2 introduced the basic concepts and techniques used to produce graphics on the TI, and even if that was all you had, it would still be a decent graphics machine. But the engineers at Texas Instruments didn't stop there. In fact, they went considerably further by giving your TI an outstanding feature: *sprite* graphics.

Sprites are graphics characters which, once defined by the program, are controlled by the TI itself. They can be made to move independently of, and concurrently with, your BASIC program. For example, your program can define a sprite in the shape of an airplane, set the plane in motion across the screen, and then go on to some other process. Meanwhile, the sprite will continue to fly across the screen until your program either interrupts it or ends.

Sprite motion is much smoother than motion controlled by BASIC. Sprites can move from one location to the next a single pixel at a time, eliminating the jumpy motion inherent in character-to-character movement. In addition, since sprites are controlled by the computer's hardware and system software, they offer a huge speed advantage over BASIC-controlled graphics. Finally, unlike other graphics, sprites can pass over other sprites or objects without erasing the previous contents of a screen location.

Sprites are placed on the screen by *pixel* location rather than character location. This means they can be placed virtually anywhere on the 192 dot-row by 256 dot-column screen matrix. The TI-99/4A allows for 32 independent sprites on the screen at one time. Extended BASIC, however, limits this to 28. Each of the 28 sprites may be assigned its own color, and any sprite's location, motion, velocity, shape, and color can be changed dynamically within a BASIC program.

There are specialized commands which let you check the status of sprites. You can, for example, determine a sprite's location and decide whether or not it is *coincident* with another sprite. Your program can also determine how far one sprite is from another sprite or screen location. The magic behind such capabilities is TI's sophisticated TMS9918A Video

Display Processor (VDP). Although several other home computers have sprite-type graphics, they do not have the automatic motion and other features of the TI because they do not have the TI chip.

The TMS9918A Video Display Processor

To fully appreciate this chip's capabilities and to better understand TI sprites, it is beneficial to take a look at the TMS9918A Video Display Processor (VDP). The video for the TI-99/4A is controlled by this chip, which is an advanced large-scale integrated-circuit (LSI). Among the features provided by the VDP are resolution of 256 x 192 pixels, 16 colors (including transparent), 35 display planes for graphics control, and sprites.

The VDP arranges the 99/4A's display into 35 geometric planes (Figure 3-1) stacked on top of each other. These stacked planes are displayed on the monitor as one composite image. Any image on the first plane will block out that area on planes 2-35, any image on the second plane will block out that area on planes 3-35, and so on. Furthermore, if the image on a particular plane is moving, it will appear to pass in front of any images on planes in front of it. The overall effect is to simulate a three-dimensional graphics display on the two-dimensional screen of your monitor.

The first 32 planes, numbered 0-31, may contain one sprite graphics character each. The sprite on the lowest-numbered plane will always appear to pass over the other sprites. The highest-numbered sprite will always appear to pass behind the other sprites. The remaining sprites act according to their precedence. Keep in mind that Extended BASIC allows you to define only 28 sprites (numbered 1-28).

Behind the sprite planes is the pattern plane. This plane contains the characters placed on the screen by BASIC commands such as DISPLAY and HCHAR. Behind this is the backdrop plane, which appears as the display area background and border on your TV or monitor. Its color is set by the CALL SCREEN command.

The last plane in the stack is the external video plane. This plane allows the VDP to mix external video signals into the image. It is used only in specialized applications and has a default color of black. It is this plane that causes the screen to black out when you set the screen color to transparent.

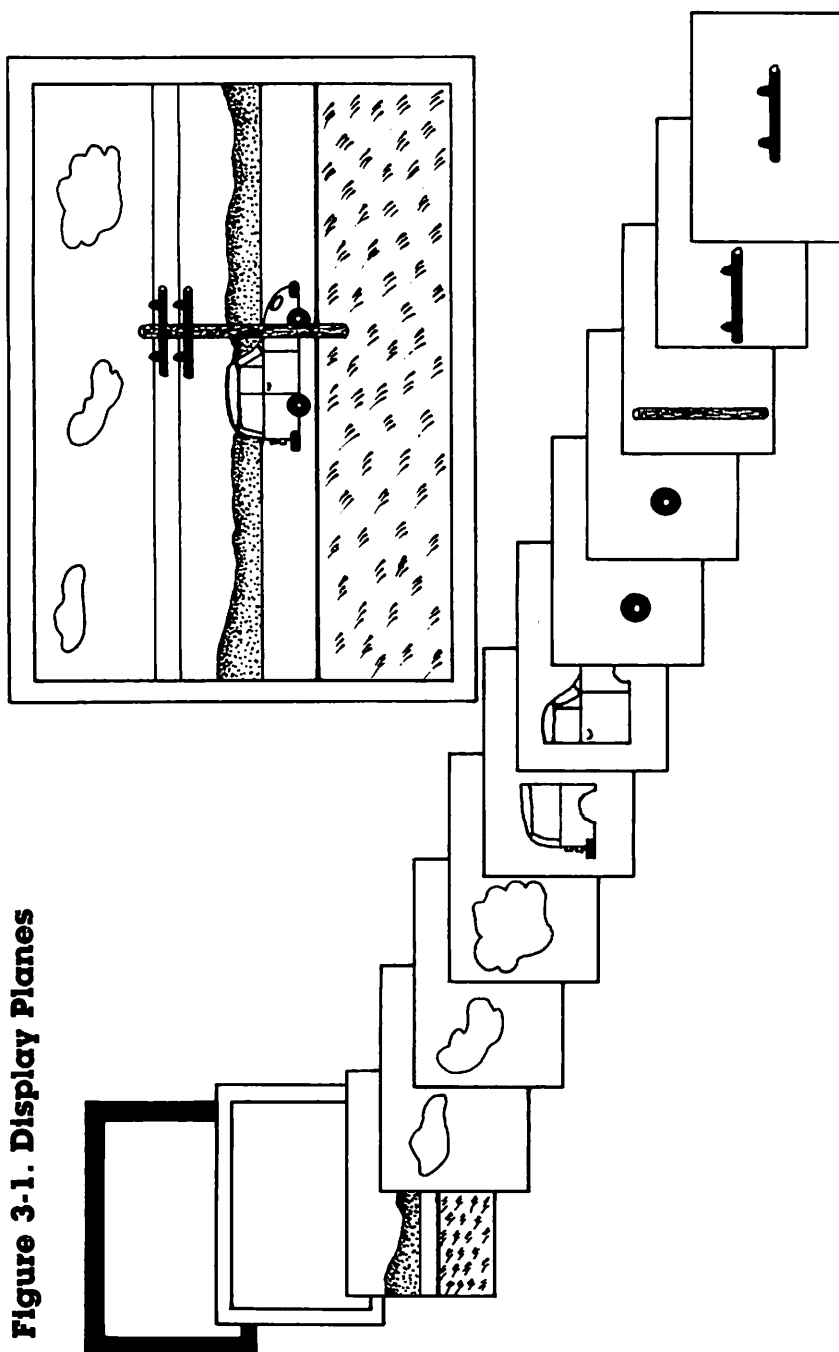


Figure 3-1. Display Planes

VDP RAM

The actual patterns, sprite attributes, and other information required to produce a screen display are stored in a block of memory called VDP Random Access Memory (RAM). This VDP RAM is segregated into sub-blocks containing various tables that define the screen image. Five of the most important sub-blocks are the Pattern Name Table, Pattern Generator Table, Pattern Color Table, Sprite Attribute Table, and Sprite Generator Table.

The first three tables define characters on the pattern plane. The Pattern Name Table identifies each of the 768 character positions on the screen and points to the pattern definition for that character position in the Pattern Generator Table. The Pattern Generator Table contains the character patterns defined to the computer which can be displayed on the screen at the locations determined by the Pattern Name Table. The Pattern Color Table contains the color codes for the patterns defined in the Pattern Generator Table, and colors are assigned to character sets (every eight contiguous characters) rather than to individual characters.

Two separate tables are used to define and control sprites. The Sprite Attribute Table identifies each sprite that has been defined and contains information about its location and color. It also contains a pointer to the sprite's entry in the Sprite Generator Table. The Sprite Generator Table defines the pattern used for that sprite.

Putting It on the Screen

So the VDP RAM contains a lot of information. How is that information used to place an image on the screen? Actually, the screen image is constantly updated by the computer, which continually scans the tables to determine what character or sprite belongs in each screen position. Information about each character or sprite (for example, color, pattern, size, etc.) is collected from the tables and mapped onto the appropriate plane, and the planes are then combined to form the actual screen image (Figure 3-2).

Obviously, since the display appears to change constantly, the process is being accomplished at microsecond speeds. VDP RAM values are being changed (via BASIC instructions); patterns, colors, and locations are being mapped onto display planes; and display planes are being combined to form the

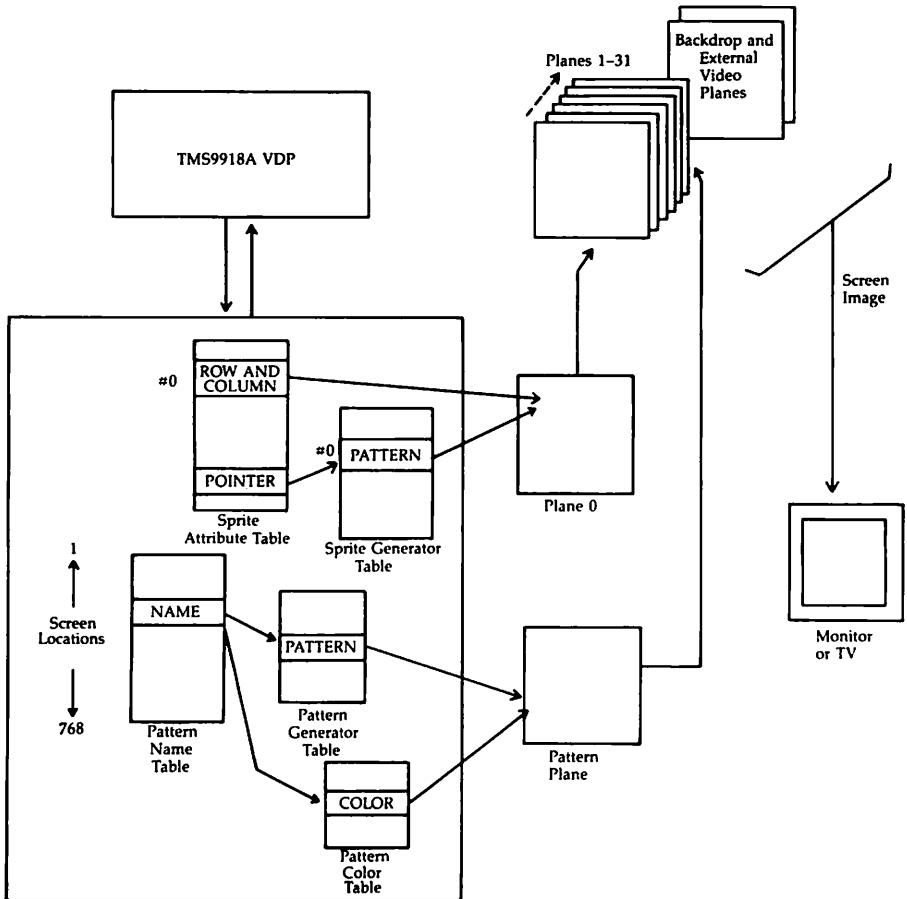
Figure 3-2. Mapping the Display

image on your monitor. It is a complex process, and it goes on continuously.

In addition, the VDP also keeps track of other information. It follows the velocity and direction of any sprites that

are in motion, and it checks sprite size, sprite coincidence (two or more sprites in the same display area), sprite distance from other objects, and the number of sprites on a line.

Defining Sprites

Sprites are defined by using the CALL SPRITE subprogram. It has the following format:

```
CALL SPRITE(#sprite number,character value,sprite color,dot-  
row,dot-column,row velocity,column velocity,...)
```

Sprite number is a numeric value from 1 to 28 which identifies the sprite. The numeric value is always preceded by the # symbol and may be a numeric literal, variable, or expression (for example, #2, #A, #X+1). If a sprite is defined with a sprite number that already exists, the old sprite is deleted and the new one takes its place.

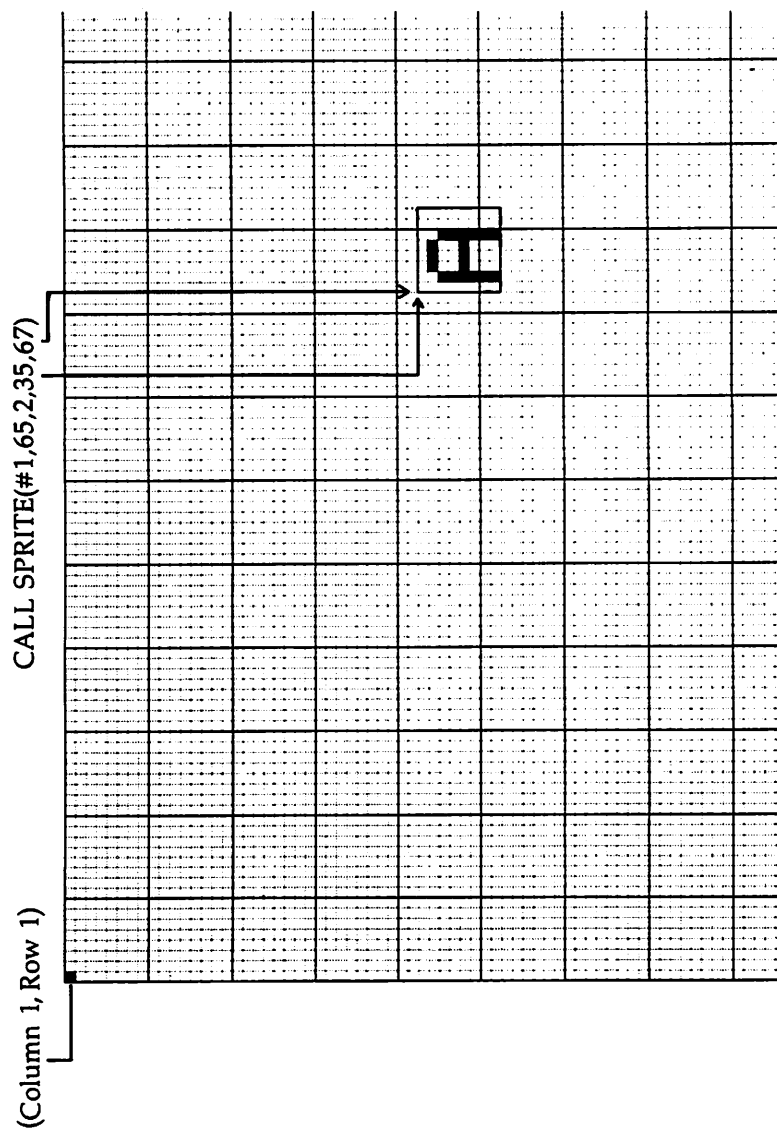
When defining sprites in your program, keep in mind that they will appear on sprite planes. Consequently, lower-numbered sprites will pass over higher-numbered ones, and all sprites will pass over other objects which are on the pattern plane.

Character value identifies the ASCII character code that contains the desired sprite pattern. The pattern is defined by the CALL CHAR subprogram. Character value must be an integer between 32 and 143, and it may be a numeric literal, numeric variable, or numeric expression. For *double-sized* sprites (discussed later), character value is the first ASCII code of a four-code contiguous group.

Sprite color is an integer from 1 to 16 that determines the sprite's color. The value itself may be a numeric literal, numeric variable, or numeric expression. The colors are the same as those used in the CALL COLOR and CALL SCREEN commands. Sprite color identifies the foreground color only; the background color is always transparent (1).

Dot row identifies a sprite's initial row location on the screen, which is determined by pixel rather than character location. As a result, a sprite can be positioned anywhere on the TI's 192 dot row screen grid. The sprite is placed on the screen with its top left pixel at the location indicated by dot-row (Figure 3-3).

Figure 3-3. Sprite Location

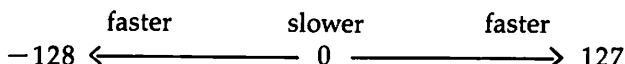


Sprites

Dot row is an integer value in the range 1–256. It may be a numeric literal, numeric variable, or numeric expression. The upper limit is 256 instead of 192 because, although the TI has the ability to *address* up to 256 rows, only the first 192 are visible. The remainder (193–256) are off the bottom of the screen, but sprites can still be located there.

Dot column identifies a sprite's initial column location on the screen. The location may be any of the 256 columns on the TI screen grid. As with dot-row position, a sprite is located at the dot-column indicated by its top-left pixel.

Row velocity is an integer in the range –128 to 127 and may be a numeric literal, numeric variable, or numeric expression. It causes the sprite to have vertical motion. When row velocity is positive, the sprite will move *downward*; when row velocity is negative, the sprite will move *upward*. The speed at which the sprite moves is determined by the row velocity value. The higher the positive number, or the lower the negative number, the faster the speed. The following scale illustrates this:



Column velocity is an integer in the range –128 to 127 and may be a numeric literal, numeric variable, or numeric expression. It causes the sprite to have horizontal motion. A positive value moves the sprite to the right; a negative value moves the sprite to the left. The sprite's speed is determined the same way as row velocity.

You can give a sprite diagonal motion by using both row and column velocity. For example, a row velocity of 20 and column velocity of 20 will cause the sprite to move down and to the right at a 45-degree angle. Other values will produce different directions and speeds. If you want the sprite to be stationary, you can either give it a row and column velocity of 0 or exclude these two parameters altogether.

There are several other factors you should consider when using sprites in your programs. Whenever a program hits a breakpoint or you use CLEAR (FCTN 4) to stop the program,

all active sprites cease to exist. The CONTINUE statement will not restore them. Also, the VDP is designed to handle a maximum of four sprites on any one screen line. When this limit is exceeded, the highest numbered sprites "disappear." They still exist, however, and will reappear once they (or one of the other sprites) move beyond the line.

CALL SPRITE Subprogram Examples

The best way to understand sprites is to see them in action. The following short programs illustrate several variations on the CALL SPRITE command. You should type in and RUN these programs to know how the commands work.

```
100 CALL CLEAR
110 CALL SPRITE(#1,65,2,95,128)
120 GOTO 120
```

This program defines sprite number 1. The pattern for the sprite is the normal pattern for ASCII code 65—the letter A. The foreground color is black (2); the background is always transparent. The sprite will be displayed on the screen with its top left-hand corner pixel at dot-row 95 and dot-column 128.

As you can see when you RUN the program, the sprite remains in the center of the screen. In order to put the sprite in motion, it must be given row and/or column velocity. Changing line 110 to the following does just that:

```
110 CALL SPRITE(#1,65,2,95,128,20,0)
```

The CALL SPRITE subprogram has now been given two additional parameters. Its row velocity is now 20; the column velocity is 0. The sprite will move smoothly downward at medium speed. When it gets to the bottom of the screen, it will wrap around the screen and reappear at the top, still in motion. The motion will continue until you end (CLEAR) the program.

If you change the row velocity to 90, the sprite's speed will show a dramatic increase. Similarly, by changing the row velocity to -20, you will make the sprite reverse direction and move upward. Changing the velocity to -90 will move the sprite upward at high speed.

Horizontal motion is controlled in the same way. When you change the row velocity back to 0 and the column velocity to 20, the sprite will move across the screen from left to

right at medium speed. Changing the column velocity to -20 moves the sprite from right to left.

By providing both row and column velocities, you can move the sprite in any direction. For example, a row velocity of 20 and a column velocity of 20 move the sprite down to the right. Similarly, row velocity 20 and column velocity -20 move the sprite down and to the left, and -10 and -50 move the sprite up and to the left at a sharp angle.

Program 3-1 demonstrates three additional concepts relating to CALL SPRITE. First, there may be multiple sprites—as many as 28—on the screen at any one time. Second, the parameters coded in the CALL SPRITE command may be generated randomly. And finally, when a sprite number is reused in a CALL SPRITE command, the sprite previously identified by that number will be replaced by the new sprite indicated in the command.

Program 3-1. Sprite Example 1

```
100 CALL CLEAR
110 CALL SCREFN(2)
120 RANDOMIZE
130 FOR L=1 TO 26
140 ROWVEL=1+INT(RND*50)
150 IF RND<.5 THEN ROWVEL=ROWVEL*-1
160 COLVEL=1+INT(RND*50)
170 IF RND<.5 THEN COLVEL=COLVEL*-1
180 COLOR=3+INT(RND*14)
190 CALL SPRITE(#L,L+64,COLOR,95,128,ROWVEL,COLVEL)
200 NEXT L
210 GOTO 130
```

Lines 100–110 clear the screen and set the color to black (2). Line 120 seeds the random number generator. Line 130 sets up a loop which will repeat 26 times. Lines 140–170 randomly generate the row velocity and column velocity for each 26 sprites. The sprite number is determined by L (1–26); the sprite pattern is determined by L+64 (which yields 65–90, representing the letters A–Z).

The sprites will appear initially at dot row 95 and dot column 128, with velocities determined by lines 140–170. Line 200 closes the loop. Line 210 branches to line 130, which starts the process over again. As each sprite number (1–26) is

reused, the sprite previously identified by a particular number is replaced on the screen by the new sprite now identified by that number.

The example programs presented thus far have used letters of the alphabet for the sprite shapes. Sprites may, of course, be any shape you define them to be. Program 3-2 combines sprites with character definition techniques discussed in Chapter 2.

Program 3-2. Sprite Example 2

```

100 CALL CLEAR
110 CALL CHAR(33,"18182424424281FF")
120 CALL CHAR(34,"FF7E3C18183C7EFF")
130 CALL CHAR(35,"183C7EFFFF7E3C18")
140 CALL CHAR(36,"80E0F0FFFFFF0E080")
150 CALL CHAR(37,"182442FFFF422418")
160 CALL SCREEN(2)
170 RANDOMIZE
180 FOR L=1 TO 26
190 ROWVEL=1+INT(RND*30)
200 IF RND<.5 THEN ROWVEL=ROWVEL*-1
210 COLVEL=1+INT(RND*30)
220 IF RND<.5 THEN COLVEL=COLVEL*-1
230 COLOR=3+INT(RND*14)
240 PAT=33+INT(RND*5)
250 CALL SPRITE(#L,PAT,COLOR,95,128,ROWVEL,COLVEL)
260 NEXT L
270 GOTO 180

```

Line 100 clears the screen. Lines 110–150 define the character patterns used for the sprites. These will be assigned to ASCII codes 33–37. Line 160 sets the screen to black, and line 170 seeds the random number generator. Line 180 sets up the sprite definition loop, which is closed by line 260. Lines 190–220 randomly generate the row and column velocities. Line 230 randomly selects the sprite color. Line 240 determines the pattern to be used for each sprite by selecting a random number between 33 and 37 inclusive, and line 250 defines the sprite and sets it in motion. Line 270 starts the process over.

Sprite Sizes

Sprites displayed on the screen are not limited in size to a single 8 x 8 character. There are, in fact, four different sprite sizes

to choose from. These are formed by specifying the sprite as either single- or double-sized and magnified or unmagnified.

Single-sized sprites are defined by a single character pattern. *Double-sized* sprites, on the other hand, are defined by four contiguous character patterns (for example, ASCII codes 100–103). The first pattern in the four-pattern group is defined by an ASCII code evenly divisible by four (for example, 100). If you specify a code that is not divisible by four, the computer will start with the next lowest ASCII code which is divisible by four. For example, if you specify ASCII code 102 as the CALL SPRITE character code, the four-pattern sprite will still be defined by ASCII codes 100–103.

Each character in an *unmagnified* sprite occupies the standard 8 x 8 character grid. The overall size of a *magnified* sprite character is increased by a factor of four. A single-sized magnified sprite occupies a 4-character 16 x 16 grid; a double-sized magnified sprite occupies a 16-character 32 x 32 grid.

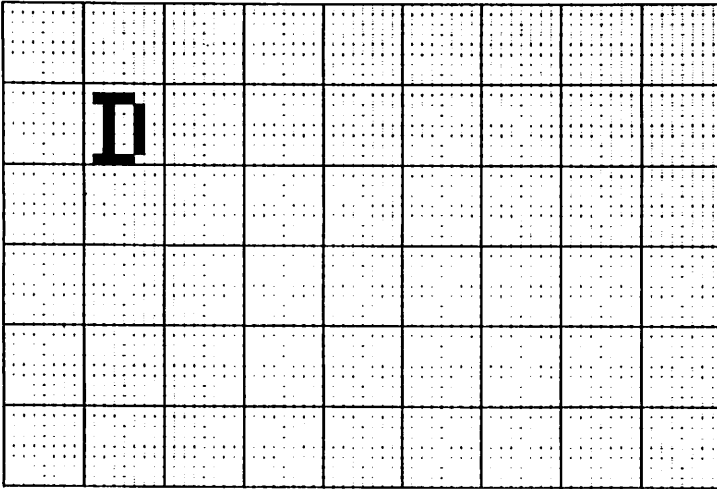
The CALL MAGNIFY subprogram is used to set the desired sprite size. It has the following format:

CALL MAGNIFY(magnification factor)

Magnification factor is an integer from 1 to 4 that defines the sprite's size. It can be a numeric literal, numeric variable, or numeric expression. The size indicated by magnification factor is assigned to all active sprites on the screen.

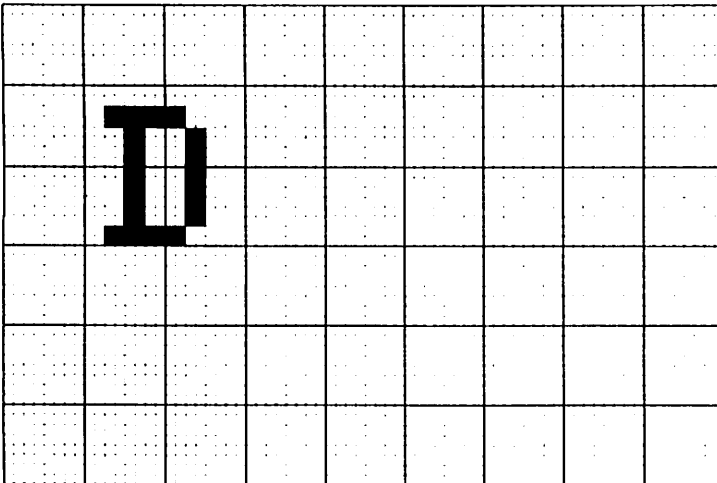
A magnification factor of 1 defines all active sprites as single-sized and unmagnified. This is the default size if no CALL MAGNIFY is specified in the program and is shown by the following program:

```
100 CALL CLEAR
110 CALL MAGNIFY(1)
120 CALL SPRITE(#1,68,2,9,9)
130 GOTO 130
```



A magnification factor of 2, as the next program shows, defines all active sprites as single-sized and magnified. The sprite pattern is displayed on the screen four times larger than normal.

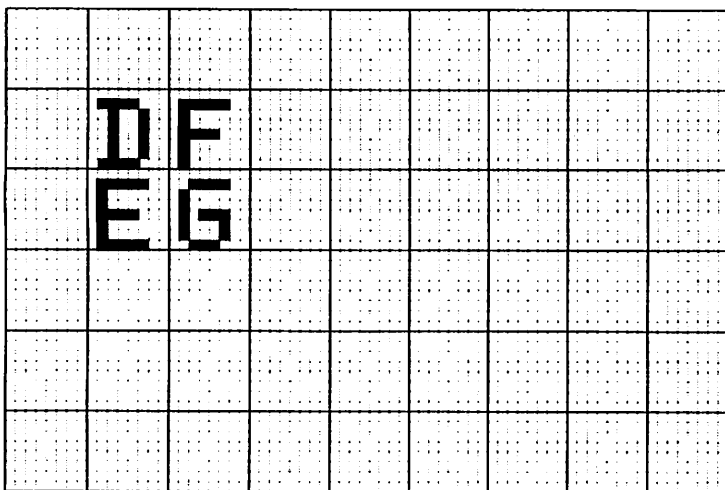
```
100 CALL CLEAR
110 CALL MAGNIFY(2)
120 CALL SPRITE(#1,68,2,9,9)
130 GOTO 130
```



Sprites

A magnification factor of 3 defines all active sprites as double-sized and unmagnified. The sprite pattern is defined by four contiguous characters. The first character code in the four-character group is evenly divisible by four. Each of the four characters is unmagnified (normal size).

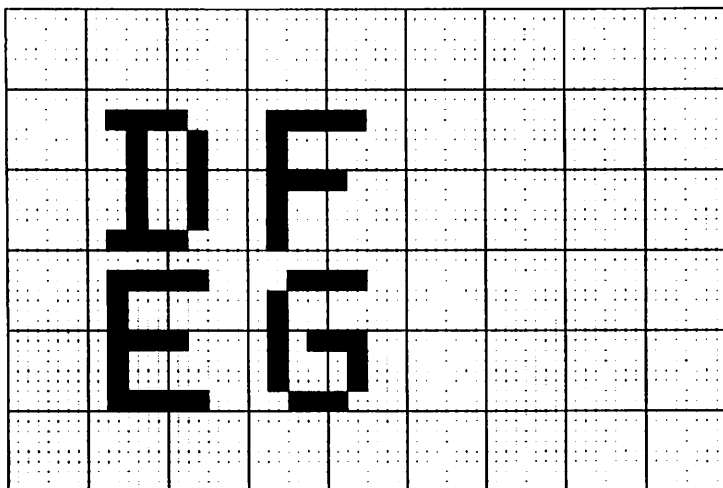
```
100 CALL CLEAR
110 CALL MAGNIFY(3)
120 CALL SPRITE(#1,68,2,9,9)
130 GOTO 130
```



Notice that the above example displays the four letters D, E, F, and G, the four-character group. If you change the character code to 70 (not evenly divisible by 4), the computer uses the next lowest number divisible by 4, which is 68. As a result, the same four letters would be displayed.

Finally, a magnification factor of 4 defines all active sprites as double-sized and magnified. The sprite pattern is defined by four contiguous characters. In addition, each of the four characters is four times its normal size.

```
100 CALL CLEAR
110 CALL MAGNIFY(4)
120 CALL SPRITE(#1,68,2,9,9)
130 GOTO 130
```



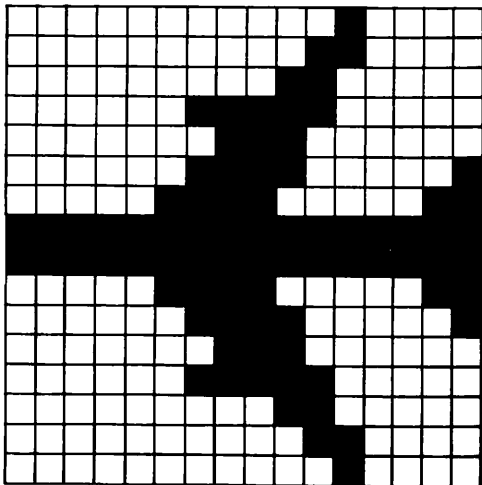
If you look at the two double-sized examples, you will notice that the four characters making up the sprite are placed on the screen in a specific order. The first ASCII character forms the top left portion of the sprite. The second forms the bottom left, the third forms the top right, and the fourth forms the bottom right.

For simplicity's sake, the preceding examples used letters of the alphabet to define the sprite patterns. However, you can define whatever patterns you wish to use by using CALL CHAR.

The short program below shows how that can be done. It uses CALL CHAR to define a pattern in four contiguous ASCII character codes. It then defines a double-sized unmagnified sprite which moves across the screen.

```

100 CALL CLEAR
110 CALL CHAR(100,"00000003010307FF")
120 CALL CHAR(101,"FF07030103000000")
130 CALL CHAR(102,"103060E0C0C183FF")
140 CALL CHAT(103,"FF83C1C0E0603010")
150 CALL MAGNIFY(3)
160 CALL SPRITE(#1,100,2,90,220,0,-10)
170 GOTO 170
    
```

Deleting Sprites

To remove a particular sprite from the screen, you must use the CALL DELSPRITE subprogram. It has the following format:

```
CALL DELSPRITE(#sprite number,...)
```

Sprite number is the number of the sprite you want to delete. It is always preceded by the # symbol and may be a numeric literal, numeric variable, or numeric expression. You may delete more than one sprite in a single CALL DELSPRITE subprogram as follows:

```
CALL DELSPRITE(#1,#D,#A+2)
```

In addition, you may delete all sprites in the program by using CALL DELSPRITE(ALL).

Sprite Color

A sprite's color may be changed by using the CALL COLOR subprogram. This command has the following format:

```
CALL COLOR(#sprite number,foreground color,...)
```

Sprite number identifies the sprite. It is always preceded by the # symbol and may be a numeric literal, numeric variable, or numeric expression. You may assign a color to more than one sprite in a single CALL COLOR command.

Foreground color is the color that will be assigned to the illuminated pixels in the sprite. A sprite's background color is always transparent.

One way to get multicolor graphics on your TI is to combine sprites of different colors. Keep in mind that sprites reside on planes, and remember that lower-numbered sprites pass in front of higher-numbered sprites when two or more are coincident (at the same location). However, only that portion of the higher-numbered sprite that is behind the foreground color of the lower-numbered sprite is actually blocked out. The portion that is behind the background color (transparent) still appears on the screen.

The following short program illustrates this. It defines two sprites. One is a solid square; the other is a triangle. When the two occupy the same location, you can see that only the portion of the square directly behind the triangle is blocked out.

```

100 CALL CLEAR
110 CALL CHAR(96,"FFFFFFFFFFFFFFFF")
120 CALL CHAR(97,"0000183C7EFF0000")
130 CALL MAGNIFY(2)
140 CALL SPRITE(#1,97,7,90,90,0,3,#2,96,3,90,140,0,-3)
150 FOR L=1 TO 1400 :: NEXT L
160 GOTO 140

```

Program 3-3, "Kaleidoscope," shows how this concept can be used to generate impressive multicolor graphics. It randomly displays four moving patterns (out of a possible six). Each pattern is assigned a randomly selected color, and the patterns and colors blend in a way that simulates a kaleidoscope.

How Kaleidoscope Works

Line(s)

- | | |
|---------|--|
| 100 | Clear the screen. |
| 110 | Seed the random number generator. |
| 120 | Set the letters of the alphabet to white. |
| 130 | Set the screen color to black. |
| 140-190 | Define the six patterns that will be used in the program. The patterns are simply symmetrical shapes that, when combined, will form a kaleidoscope effect. The sprites will have a magnification factor of 4, which means each pattern defines four contiguous characters. |

Sprites

- Notice that the ASCII codes used to define the patterns are all evenly divisible by four.
- 200 Load the C array with the six color codes that will be used by the sprites.
- 210 Load the CS array with the ASCII codes which define the six patterns.
- 220 Display the program title.
- 230 Set the sprite magnification factor to 4 (double-sized and magnified).
- 240 Randomly select one of the six character patterns and one of the six colors.
- 250 Define the sprite with the lowest precedence (#4). The pattern and color come from the random selection in lines 200-210.
- 260-320 Define the remaining three sprites. The character pattern and color are randomly selected. All four sprites have the same initial screen location (row 90, column 100). Lines 270, 300, and 330 are timing loops.
- 350 Start the process over. The overall effect is a continually changing kaleidoscope of patterns and colors.

Program 3-3. Kaleidoscope

```
100 CALL CLEAR
110 RANDOMIZE
120 FOR L=5 TO 8 :: CALL COLOR(L,16,1):: NEXT L
130 CALL SCREEN(2)
140 CALL CHAR(96,"FF80809F98949392939394989F8080FF
FF0101F91929C94949C92919F90101FF")
150 CALL CHAR(104,"3E2AD3A2CF8EFC2121FC8ECEA2D32A3
E7C54CB4573713F84843F717345CB5'47C")
160 CALL CHAR(112,"FE8080838084921111928480838080F
E7F0101C10121498888492101C1010170")
170 CALL CHAR(120,"FFECDB0E0C784858584C7E0B0D8ECF
FFF371B0D07E321A1A121E3070D1B37FF")
180 CALL CHAR(128,"FFC0A093898492999992848993A0C0F
FFF0305C99121499999492191C90503FF")
190 CALL CHAR(136,"3F4F8783C1E0F0F8F8F0E0C183874F3
FFCF2E1C183070F1F1F0F0783C1E1F2FC")
200 C(1)=13 :: C(2)=5 :: C(3)=9 :: C(4)=12 :: C(5)
=14 :: C(6)=16
210 CS(1)=96 :: CS(2)=104 :: CS(3)=112 :: CS(4)=12
0 :: CS(5)=128 :: CS(6)=136
220 DISPLAY AT(4,3):"K A L E I D O S C O P E"
230 CALL MAGNIFY(4)
```

```

240 S4=1+INT(RND*6):: C4=1+INT(RND*6)
250 CALL SPRITE(#4,CS(S4),C(C4),90,100)
260 S3=1+INT(RND*6):: C3=1+INT(RND*6)
270 FOR L=1 TO 25 :: NEXT L
280 CALL SPRITE(#3,CS(S3),C(C3),90,100)
290 S2=1+INT(RND*6):: C2=1+INT(RND*6)
300 FOR L=1 TO 25 :: NEXT L
310 CALL SPRITE(#2,CS(S2),C(C2),90,100)
320 S1=1+INT(RND*6):: C1=1+INT(RND*6)
330 FOR L=1 TO 25 :: NEXT L
340 CALL SPRITE(#1,CS(S1),C(C1),90,100)
350 GOTO 240

```

Controlling Sprites in BASIC Programs

In order for sprites to be truly useful in your programs, you need a way to control them. This is particularly true for games, when the person at the keyboard must be able to control a sprite's location, speed, or direction.

One way to do this is to have the program redo the CALL SPRITE subprogram each time a change is required. This was done in the previous program, "Kaleidoscope." Usually, however, it is not necessary to completely redefine a sprite. Extended BASIC provides several subprogram commands that let you change only a specific parameter.

Changing a Sprite's Location

A sprite's location on the screen can be changed by using the CALL LOCATE subprogram. All of the sprite's other attributes are left intact.

CALL LOCATE(#sprite number, dot-row, dot-column,...)

Sprite number identifies the sprite that will have its location changed. It is an integer in the range 1-28 and may be a numeric literal, numeric variable, or numeric expression. It is always preceded by the # symbol.

Dot-row identifies the new row location for the sprite. It is an integer in the range 1-256 and may be a numeric literal, numeric variable, or numeric expression. Dot-column identifies the new column location for the sprite and is also an integer in the range 1-256.

The following short program illustrates how CALL LOCATE works. It defines a sprite in the shape of an A and then randomly locates it on the screen.

```

100 CALL CLEAR
110 CALL SPRITE(#1,65,2,90,125)

```

```
120 FOR L=1 to 500 :: NEXT L
130 CALL LOCATE(#1,1+INT(RND*192),INT(RND*250))
140 GOTO 120
```

Changing a Sprite's Motion

- A sprite's direction and speed can be changed using the CALL MOTION subprogram. All of the sprite's other attributes are left intact. The subprogram has the following format:

CALL MOTION(#sprite number, row velocity, column velocity,...)

Sprite number identifies the sprite. It is an integer in the range 1–28 and may be a numeric literal, numeric variable, or numeric expression. It is always preceded by the # symbol.

Row velocity determines the sprite's vertical speed. It is an integer in the range –128 to 127. A positive value moves the sprite downward; a negative value moves the sprite upward. The sprite's speed is determined by how far the row-velocity value is from 0. That is, as the value increases from 0 to 127 or decreases from 0 to –128, the speed increases. A value of 0 indicates no vertical motion.

Column velocity determines the sprite's horizontal speed and is also an integer in the range –128 to 127. A positive value moves the sprite to the right; a negative value moves the sprite to the left. Horizontal speed increases as the value of column velocity gets further from 0.

Diagonal motion is achieved by using combinations of row and column velocities. For example, a row velocity of 20 and column velocity of 20 moves the sprite down and to the right.

The following short program illustrates how CALL MOTION works. It creates a sprite in the middle of the screen and then moves it around randomly by changing the row and column velocities.

```
100 CALL CLEAR
110 RANDOMIZE
120 CALL CHAR(96,"81423C18183C4281")
130 DISPLAY AT(2,10):"DOODLEBUG"
140 CALL SPRITE(#1,96,2,90,125)
150 A=INT(RND*20)-INT(RND*20)
160 B=INT(RND*20)-INT(RND*20)
170 CALL MOTION(#1,A,B)
180 GOTO 150
```

Line 100 clears the screen, and 110 seeds the random number generator. Line 120 defines the sprite pattern. Line 140 creates the sprite in the middle of the screen. Lines 150–160 generate row and column velocities between -19 and 19 . Line 170 issues the CALL MOTION subprogram for sprite #1 using the row and column velocities obtained randomly in lines 150–160. Line 180 puts the program in an endless loop.

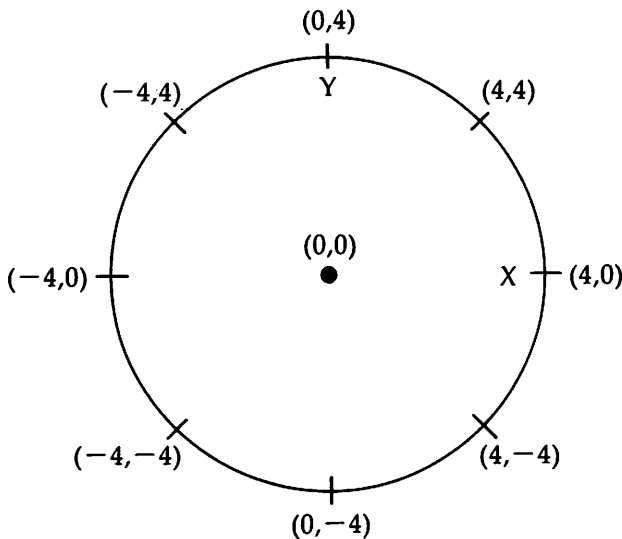
Controlling Sprites with Joysticks

Joysticks provide one means of communicating with your TI. They allow you to move objects, control motion, and fire lasers. Of course, the program must be designed to acknowledge the instructions coming from the joysticks. This is done by using the CALL JOYST subprogram.

CALL JOYST(key unit, X-return, Y-return)

When your program encounters a CALL JOYST statement, the TI queries the joystick port to determine the direction in which the stick is pushed. *Key unit* is the number of the joystick being tested. It is an integer with a value of 1 or 2, corresponding to joystick 1 or joystick 2.

Figure 3-4. CALL JOYST Return Values



The result of the query is returned to your program as values for the two variables identified by *X-return* and *Y-return*. The values returned of variables indicate the direction in which the stick was pushed. The following short program illustrates how the values may be used to control motion. It uses the joystick to move a crosshair sight around the screen. The ALPHA LOCK key must be up.

```
100 CALL CLEAR
110 CALL CHAR(96,"181818FFFF181818")
120 CALL MAGNIFY(2)
130 CALL SPRITE(#1,96,2,90,125)
140 CALL JOYST(1,X,Y)
150 CALL MOTION(#1,-Y,X)
160 GOTO 140
```

Line 100 clears the screen. Line 110 defines the crosshair pattern. Line 120 sets a magnification factor of 2, and line 130 defines the sprite on the screen. Line 140 queries the joystick and places the returned values in X and Y. Line 150 then moves the sprite based on the values of X and Y, and line 160 initiates a loop to continue the process.

If the joystick were pushed to the left, the returned values would be $X = -4$ and $Y = 0$. The row velocity in the CALL MOTION command is determined by $-Y$, which results in -0 or just 0. The column velocity is determined by X, which is -4 . A row velocity of 0 and column velocity of -4 moves the sprite (crosshair) to the left.

The other joystick positions will return different values to produce an appropriate effect on the sprite. When the joystick is not pushed (0,0), the sprite will be stationary.

You can increase or decrease the sprite's speed by applying a factor to the CALL MOTION velocities. For example, if you wanted the sprite to move four times faster, you could use the following line:

```
CALL MOTION(#1,-4*Y,4*X)
```

Controlling Sprites from the Keyboard

There are situations in which you might want to use the TI console keyboard, instead of joysticks, to control the action in a program. For instance, a program might require more information than just direction. Or, for that matter, you might not have any joysticks. In such cases, the CALL KEY subprogram

lets you use the keyboard to control the program action.

CALL KEY(key unit,return variable,status code)

When a CALL KEY subprogram command is encountered in a program, the computer queries the keyboard to determine if a key was pressed. If one was pressed, a value returned by the subprogram identifies the key. The value returned depends on the *key unit*.

When key-unit is 0, the *return variable* value is the same as the ASCII character code for that key, including lowercase letters. In other words, pressing SHIFT-A would return a value of 65, while pressing an unshifted A would return a value of 97. When key-unit is 3, the return variable value is the same as the uppercase ASCII character code for that key. In this instance, pressing either a shifted or unshifted A would return a value of 65. In both cases, the full keyboard can be used.

When key unit is 1 or 2, the console is placed into the split-keyboard mode. Key-unit 1 accepts input from the left side of the keyboard, while key-unit 2 accepts input from the right side. This allows each player in two-player games to have his or her own set of control keys.

When the console is in split-keyboard mode, the return variable is not the same as the ASCII code for that key. Rather, the value returned is one of those listed in Table 3-1.

Status code indicates whether or not a key was pressed. A status of 0 indicates that no key was pressed. A status of -1 indicates that the key pressed was the same as the last one pressed; a status of 1 indicates a new key was pressed.

The following program illustrates how CALL KEY can be used to control the action. It moves a crosshair around the screen by using the E, X, S, and D keys in full-keyboard mode. The ALPHA LOCK key must be depressed.

```

100 CALL CLEAR
110 CALL CHAR(96,"181818FFFF181818")
120 CALL MAGNIFY(2)
130 CALL SPRITE(#1,96,2,90,125)
140 CALL KEY(0,K,S)
150 Y=(K=88)-(K=69) :: X=(K=83)-(K=68)
160 CALL MOTION(#1,-Y,X)
170 GOTO 140

```

Line 100 clears the screen. Line 110 defines the character pattern for the crosshair. Line 120 sets a magnification factor

Table 3-1. Split Keyboard Codes

| Left | Right | Code |
|---------|-------|------|
| X | M | 0 |
| A | H | 1 |
| S | J | 2 |
| D | K | 3 |
| W | U | 4 |
| E | I | 5 |
| R | O | 6 |
| 2 | 7 | 7 |
| 3 | 8 | 8 |
| 4 | 9 | 9 |
| 5 | 0 | 10 |
| T | P | 11 |
| F | L | 12 |
| V | . | 13 |
| C | , | 14 |
| Z | N | 15 |
| SHIFT-B | / | 16 |
| G | ; | 17 |
| Q | Y | 18 |
| 1 | 6 | 19 |

of 2 for sprites. Line 130 displays the sprite on the screen, and line 140 queries the keyboard to see if a key was pressed.

Line 150 uses two logical expressions to set the values of X and Y. A logical expression returns a value of -1 for a true condition and 0 for a false condition. For example, if the S key (ASCII 83) were pressed, $K=83$ would be true, resulting in a value of -1 ; $K=68$ would be false, resulting in a value of 0. Consequently, X would be -1 ($-1 - 0$). Y would be 0 since both $K=88$ and $K=69$ are false ($0 - 0$). These calculations give the CALL MOTION statement in line 160 a row velocity of 0 and a column velocity of -1 . As a result, the crosshair moves to the left. Line 170 continues the process.

When no key is pressed, all values would equal 0 and the crosshair would be stationary. However, you could keep the crosshair, or any sprite, moving in the direction indicated by the last key pressed by adding the following line:

```
145 IF S=0 THEN 140
```

This instruction prevents a "no key pressed" condition from reaching the CALL MOTION statement.

You could, of course, use a series of IF statements in place of the logical expressions in line 150 to determine which key had been pressed. For example, the following lines would have the same result as the logical expressions:

```
150 IF K=83 THEN Y=-1
151 IF K=68 THEN Y=1
152 IF K=88 THEN X=1
153 IF K=69 THEN X=-1
```

As you can see, however, this method requires more program lines than does the use of logical expressions. It is also less efficient.

Joystick and Keyboard Control

Programs may be set up so that they can use either the joysticks or the keyboard to control the action, with the program itself determining which will be used. The following program, again using the crosshair sight example, shows how this is accomplished. The ALPHA LOCK key must be up.

```
100 CALL CLEAR
110 CALL CHAR(96,"181818FFFF181818")
120 CALL MAGNIFY(2)
130 CALL SPRITE(#1,96,2,90,125)
140 CALL KEY(3,K,S):: IF S=0 THEN CALL JOYST(1,X,Y)::
    GOTO 160
150 Y=((K=88)-(K=69))*4 :: X=((K=83)-(K=68))*4
160 CALL MOTION(#1,-Y,X)
170 GOTO 140
```

Line 100 clears the screen. Line 110 defines the sprite pattern, and line 120 gives it a magnification factor of 2. Line 130 creates the sprite on the screen. Line 140 checks to see if a key has been pressed. If not, the joystick port is checked, and the logical expressions in line 150 are bypassed. If a key was pressed, then the joystick check is bypassed, and the logical expressions in line 150 are used to calculate the values for the CALL MOTION subprogram in line 160. Line 170 continues the process. The CALL KEY subprogram in line 140 uses key-unit 3 so that the value returned will always represent an uppercase letter. This is necessary since the logical expressions in line 150 assume uppercase.

Changing Sprite Patterns

There are many times when it is useful to change a sprite's pattern in a program. For example, if a car-shaped sprite was moving from left to right and then changed direction, you would want the front of the car to be facing the new direction and would need a sprite pattern for each direction the car would face. Multiple patterns are also required whenever animation is involved. Just as cartoons are created by continually changing frames, computer animation is created by changing sprite patterns. Of course, all necessary patterns must first be defined.

A sprite's pattern can be changed by using the CALL PATTERN subprogram. It has the following format:

CALL PATTERN(#sprite number,character code)

Sprite number identifies the sprite to be changed. Character code is a numeric literal, numeric variable, or numeric expression that identifies the ASCII code to be used for the sprite's pattern. It must be an integer in the range 32–143. The desired pattern would have been defined earlier in the program by the CALL CHAR subprogram.

Program 3-4, "Birds At Night," demonstrates how CALL PATTERN can produce animation. It displays a starlit night, a full moon, and a flock of birds flying across the night sky.

How Birds at Night Works

Line(s)

- | | |
|---------|--|
| 100 | Clear the screen. |
| 110–120 | Define the patterns used to animate the birds. ASCII code 96 contains the pattern for birds with wings in the up position; code 97 contains the pattern for birds with wings in the down position. |
| 130 | Define the four characters that make up the moon. |
| 140 | Define the stars. |
| 150–180 | Randomly place 50 stars on the screen. |
| 190 | Create the four sprites which make up the moon. |
| 200 | Set the screen color to dark blue to simulate nighttime. |
| 210–240 | Generate ten sprites for the flock of birds. |
| 250–260 | Initiate a loop that repeats ten times. Each iteration assigns the wings-down pattern to one of the sprites. |

- 270-280 Initiate a loop that repeats ten times. Each iteration assigns the wings-up pattern to one of the sprites.
- 290 Produces an endless loop to keep the process going.

Program 3-4. Birds at Night

```

100 CALL CLEAR
110 CALL CHAR(96,"0041221408000000")
120 CALL CHAR(97,"0000007708000000")
130 CALL CHAR(120,"00070F1F1F3F3F3F3F1F1F0F070
    000E0F0F8F0FCFCFCFCFCFCF8F8F0E000")
140 CALL CHAR(112,"001"):: CALL COLOR(11,16,1)
150 FOR L=1 TO 50
160 A=1+INT(RND*30):: B=1+INT(RND*22)
170 CALL HCHAR(B,A,112,1)
180 NEXT L
190 CALL SPRITE(#21,120,15,30,40,#22,121,15,38,40,
    #23,122,15,30,48,#24,123,15,38,48)
200 CALL SCREEN(5)
210 FOR L=11 TO 20
220 CALL SPRITE(#L,96,2,35+INT(RND*50),240,0,-3)
230 FOR L2=1 TO INT(RND*250):: NEXT L2
240 NEXT L
250 FOR L=11 TO 20
260 CALL PATTERN(#L,97):: NEXT L
270 FOR L=11 TO 20
280 CALL PATTERN(#L,96):: NEXT L
290 GOTO 250

```

Combining Techniques

The preceding programs have served primarily to illustrate particular concepts and techniques. More elaborate programs, however, usually combine several different techniques. Program 3-5, "Dot Gobbler," is the result of one such combination.

The program displays a board covered with randomly placed dots. In the middle of the board is the Dot Gobbler, a creature that gets its nourishment by eating dots. The object of the game is to consume dots by moving the Dot Gobbler around the board with the E, X, S, and D (arrow) keys.

The Dot Gobbler eats dots as he moves across them. But he can only move in a direction that has a dot, so you'll have

to think ahead. Once the Gobbler can no longer move, the game is over. Your score is based on the number of dots eaten.

How Dot Gobbler Works

Line(s)

- 100 Clear the screen.
- 110 Seed the random number generator.
- 120–160 Define the five four-character patterns required for the Dot Gobbler. ASCII code 96 defines the Gobbler with his mouth closed; code 100 defines the Gobbler moving to the right; code 104 defines the Gobbler moving to the left; code 108 defines the Gobbler moving upward; and code 112 defines the Gobbler moving downward.
- 170 Define the border pattern for the board.
- 180 Define the dot pattern.
- 190 Draw the border.
- 200–230 Randomly place the dots on the board.
- 240 Set the Gobbler's magnification factor to 3.
- 250 Display the Gobbler (sprite) on the screen.
- 260–280 Set variables which contain the Gobbler's initial row and column position. Display instruction.
- 290 Use full keyboard to get information from console. Active keys are E, S, D, and X—the arrow keys.
- 300–330 Route the program to the appropriate logic as determined by the key pressed.
- 340 End the game and start over.
- 350 Keep checking for a pressed key.
- 360–440 Control logic for moving to the right. Line 360 determines if a move to the right is valid by checking to see if the character to the right is a dot (ASCII 116). If not, line 370 checks another key. If the character is a dot, line 380 increments the sprite-column position by 8 (one character) and moves the sprite to that location. Line 390 closes the Gobbler's mouth, and line 400 introduces a short delay. Line 410 then opens the Gobbler's mouth. Line 420 updates the Gobbler's character-column and erases the eaten dot. Line 430 updates the screen display, and line 440 checks for another key.

Sprites

```
350 GOTO 290
360 CALL GCHAR(CR,CC+1,CH)
370 IF CH<>116 THEN 290
380 SC=SC+8 :: CALL LOCATE(#1,SR,SC)
390 CALL PATTERN(#1,96)
400 FOR L=1 TO 10 :: NEXT L
410 CALL PATTERN(#1,100)
420 CC=CC+1 :: CALL HCHAR(CR,CC,32,1)
430 SCOR=SCOR+1 :: DISPLAY AT(24,16):SCOR
440 GOTO 290
450 CALL GCHAR(CR,CC-1,CH)
460 IF CH<>116 THEN 290
470 SC=SC-8 :: CALL LOCATE(#1,SR,SC)
480 CALL PATTERN(#1,96)
490 FOR L=1 TO 10 :: NEXT L
500 CALL PATTERN(#1,104)
510 CC=CC-1 :: CALL HCHAR(CR,CC,32,1)
520 SCOR=SCOR+1 :: DISPLAY AT(24,16):SCOR
530 GOTO 290
540 CALL GCHAR(CR-1,CC,CH)
550 IF CH<>116 THEN 290
560 SR=SR-8 :: CALL LOCATE(#1,SR,SC)
570 CALL PATTERN(#1,96)
580 FOR L=1 TO 10 :: NEXT L
590 CALL PATTERN(#1,108)
600 CR=CR-1 :: CALL HCHAR(CR,CC,32,1)
610 SCOR=SCOR+1 :: DISPLAY AT(24,16):SCOR
620 GOTO 290
630 CALL GCHAR(CR+1,CC,CH)
640 IF CH<>116 THEN 290
650 SR=SR+8 :: CALL LOCATE(#1,SR,SC)
660 CALL PATTERN(#1,96)
670 FOR L=1 TO 10 :: NEXT L
680 CALL PATTERN(#1,112)
690 CR=CR+1 :: CALL HCHAR(CR,CC,32,1)
700 SCOR=SCOR+1 :: DISPLAY AT(24,16):SCOR
710 GOTO 290
```

Sprite Editor

By now it should be obvious that sprites require a lot of pattern definition. Before you rush out to buy a dozen pads of graph paper, however, take a look at the next program. Called "Sprite Editor," it lets you define sprites on the screen. It will provide you with either an 8 x 8 (single-sized) or 16 x 16 (double-sized) grid; you need only to indicate which pixels should be turned on in the grid by moving the cursor and

pressing 1. When you're finished, the program will display the sprite on the screen. You can magnify it and change its color. If it doesn't look quite right, you can go back to the original grid and modify the pattern. Once you are satisfied, the computer will display the hex character pattern required by CALL CHAR.

Program 3-6. Sprite Editor

```

100 CALL CLEAR
110 CALL CHAR(96,"FF010101010101")
120 CALL CHAR(104,"FFFFFFFFFFFFFF")
130 CALL CHAR(112,"FFC3A59999A5C3FF")
140 HEX$="0123456789ABCDEF"
150 CALL COLOR(9,2,16,10,2,2)
160 DISPLAY AT(4,3): "*** SPRITE EDITOR ***"
170 DISPLAY AT(8,1): "1 - ONE CHARACTER SPRITE"
180 DISPLAY AT(10,1): "2 - FOUR CHARACTER SPRITE"
190 DISPLAY AT(14,1): "SELECT----> "
200 ACCEPT AT(14,12)VALIDATE("12")SIZE(-1)PEEP:S
210 CALL CLEAR
220 FOR L1=1 TO S*8
230 DISPLAY AT(L1+2,3):RPT$(CHR$(96),S*8)
240 NEXT L1
250 DISPLAY AT(1,1):"0=PIXEL OFF --- 1=PIXEL ON"
260 DISPLAY AT(20,1):"E = MOVE CURSOR UP"
270 DISPLAY AT(21,1):"X = MOVE CURSOR DOWN"
280 DISPLAY AT(22,1):"S = MOVE CURSOR LEFT"
290 DISPLAY AT(23,1):"D = MOVE CURSOR RIGHT"
300 DISPLAY AT(24,1):"Q = QUIT SPRITE DEFINITION"
310 ROW=17 :: COL=33 :: CR=3 :: CC=3
320 CALL SPRITE(#1,112,7,ROW,COL)
330 FOR D=1 TO 25 :: NEXT D
340 CALL KEY(3,K,ST):: IF ST=0 THEN 340
350 IF K=81 THEN 570
360 IF K=69 THEN GOSUB 430
370 IF K=88 THEN GOSUB 460
380 IF K=83 THEN GOSUB 490
390 IF K=68 THEN GOSUB 520
400 IF K=48 THEN GOSUB 550
410 IF K=49 THEN GOSUB 560
420 GOTO 320
430 IF ROW-8<17 THEN 450
440 ROW=ROW-8 :: CR=CR-1
450 RETURN
460 IF ROW+8>(17+(63*S))THEN 480
470 ROW=ROW+8 :: CR=CR+1
480 RETURN

```


Sprites

```
490 IF COL-8<33 THEN 510
500 COL=COL-8 :: CC=CC-1
510 RETURN
520 IF COL+8>(33+(63*S))THEN 540
530 COL=COL+8 :: CC=CC+1
540 RETURN
550 DISPLAY AT(CR,CC):CHR$(96);: RETURN
560 DISPLAY AT(CR,CC):CHR$(104);: RETURN
570 CALL DELSPRITE(#1)
580 GOSUB 920
590 IF S=1 THEN MG=1 ELSE MG=3
600 CALL MAGNIFY(MG)
610 CALL SPRITE(#2,120,2,40,190)
620 DISPLAY AT(19,1):"C = SET SPRITE COLOR"
630 DISPLAY AT(20,1):"M = MAGNIFY SPRITE"
640 DISPLAY AT(21,1):"P = CHANGE SPRITE PATTERN"
650 DISPLAY AT(22,1):"D = DISPLAY SPRITE PATTERN"
660 DISPLAY AT(23,1):"N = NEW SPRITE"
670 DISPLAY AT(24,1):"SELECT---> "
680 ACCEPT AT(24,1)VALIDATE("CMPDN")SIZE(-1)BEEP:
    SEL$
690 IF SEL$="C" THEN 740
700 IF SEL$="M" THEN 790
710 IF SEL$="P" THEN CALL DELSPRITE(#2):: CALL MAG
    NIFY(1):: COTO 260
720 IF SEL$="N" THEN CALL DELSPRITE(ALL):: CALL MA
    GNIFY(1):: CALL CLEAR :: GOTO 160
730 IF SEL$="D" THEN 840
740 DISPLAY AT(24,1):"SPRITE COLOR (1-16)?"
750 ACCEPT AT(24,22)VALIDATE(NUMERIC)BEEP:SC
760 IF SC<1 OR SC>16 THEN 750
770 CALL COLOR(#2,SC)
780 GOTO 620
790 IF MG=1 THEN MG=2 :: GOTO 830
800 IF MG=2 THEN MG=1 :: GOTO 830
810 IF MG=3 THEN MG=4 :: GOTO 830
820 IF MG=4 THEN MG=3 :: GOTO 830
830 CALL MAGNIFY(MG):: GOTO 620
840 DISPLAY AT(19,1):PAT$(1)
850 IF S=2 THEN 870
860 FOR L1=20 TO 24 :: DISPLAY AT(L1,1):" " :: NEX
    T L1 :: GOTO 890
870 DISPLAY AT(20,1):PAT$(2):: DISPLAY AT(21,1):PA
    T$(3):: DISPLAY AT(22,1):PAT$(4)
880 DISPLAY AT(23,1):" "
890 DISPLAY AT(24,1):"---> PRESS ANY KEY"
900 CALL KEY(3,K,ST):: IF ST=0 THEN 900
910 GOTO 620
920 FOR L=19 TO 24 :: DISPLAY AT(L,1):" " :: NEXT
    L
```

```

930 DISPLAY AT(22,5):"STAND BY..."
940 CR=3 :: CC=3 :: SB=1 :: GOSUB 1010
950 IF S=1 THEN 990
960 CR=11 :: CC=3 :: SB=2 :: GOSUB 1010
970 CR=3 :: CC=11 :: SB=3 :: GOSUB 1010
980 CR=11 :: CC=11 :: SB=4 :: GOSUB 1010
990 CALL CHAR(120,PAT$(1)):: CALL CHAR(121,PAT$(2)
   ):: CALL CHAR(122,PAT$(3)):: CALL CHAR(123,PAT
   $(4))
1000 RETURN
1010 PAT$(SB)="" :: FOR L1=CR TO CR+7
1020 FOR L2=CC TO CC+7
1030 CALL GCHAR(L1,L2+2,CH)
1040 IF CH=96 THEN BITS(L2-CC+1)=0 ELSE BITS(L2-CC
   +1)=1
1050 NEXT L2
1060 HIGH=BITS(1)*8+BITS(2)*4+BITS(3)*2+BITS(4)+1
1070 LOW=BITS(5)*8+BITS(6)*4+BITS(7)*2+BITS(8)+1
1080 PAT$(SB)=PAT$(SB)&SEG$(HEX$,HIGH,1)&SEG$(HEX$
   ,LOW,1)
1090 NEXT L1
1100 RETURN

```



Advanced Sprite-Handling Techniques

Advanced Sprite-Handling Techniques

The material in Chapter 3 examines various methods for controlling sprites from BASIC. Simply controlling sprites, however, is usually not enough to produce a complete program. There must also be some type of feedback mechanism that indicates the status of each of the sprites on the screen; such status information can then be used to make sprites interact with each other and with nonsprite graphics.

TI Extended BASIC contains three useful subprogram commands that allow your program to monitor sprites. CALL POSITION can be used to find a sprite's current location on the screen. CALL COINC can be used to determine if two sprites, or a sprite and a screen location, are coincident (at the same location). Finally, CALL DISTANCE can be used to determine how far one sprite is from another sprite or from a specified screen location.

This chapter shows you how to use these commands in BASIC programs and how to write complete programs using the various sprite subprogram commands.

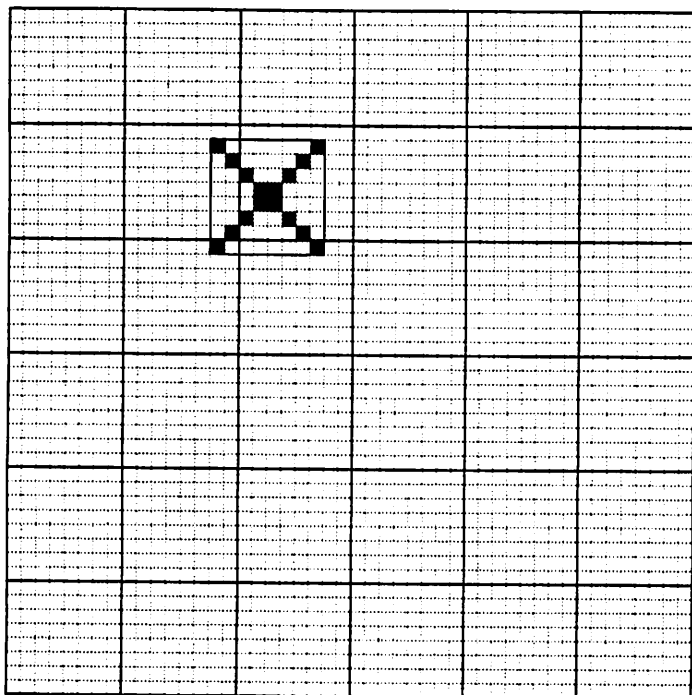
Finding a Sprite's Screen Position

The CALL POSITION subprogram is used to determine a sprite's current location on the screen. It has the following format:

CALL POSITION (#sprite number,dot row,dot column,...)

Sprite number identifies the sprite and is always preceded by the # symbol. It is an integer in the range 1-28 and may be a numeric literal, numeric variable, or numeric expression. The row and column locations of the sprite are returned to the program in the two integer variables dot row and dot column. Dot row identifies one of the 192 possible row positions (actually 256 counting the "invisible" rows); dot column identifies one of the 256 possible column positions.

In Figure 4-1, for instance, the CALL POSITION subprogram would return 10 for the value of R and 15 for the value of C. Notice that the locations returned are defined by the top left pixel in the sprite. If the sprite identified by that particular sprite number did not exist, the values would be 0,0.

Figure 4-1. Sprite at Row 10 and Column 15

The values returned by the CALL POSITION subprogram are determined by the location of the sprite at the time the command is issued. Consequently, the values returned for a moving sprite are only valid for a short period of time (that is, while the sprite is still in that location). The values will not be updated until another CALL POSITION subprogram command is encountered.

The following short program lets you move a sprite around the screen by using the E, S, D, and X keys. The sprite's location, as determined by the CALL POSITION subprogram, is displayed at the bottom of the screen.

```
100 CALL CLEAR
110 CALL SPRITE(#1,65,2,90,125)
120 CALL KEY(3,K,S)
130 X=(K=83)-(K=68) :: Y=(K=88)-(K=69)
```

Advanced Sprite Handling Techniques

```
140 CALL MOTION(#1,-Y,X)
150 CALL POSITION(#1,R,C)
160 DISPLAY AT (23,2):"POSITION: ROW";R;"COL";C
170 GOTO 120
```

The information returned by CALL POSITION can be very useful in your BASIC programs. You can, for example, use the values to determine if a sprite has reached a particular location on the screen. You can also prevent sprites from wrapping around the screen by continually checking their position and then stopping, or changing, their motion when they reach the edge.

You can also use the values returned by the CALL POSITION subprogram to have one sprite shoot at or chase another sprite. Keep in mind that the dot row and dot column values simply identify a sprite's location on the 256 x 192 screen grid. By identifying the location of both sprites, a simple formula can be used to calculate the values required by the CALL MOTION subprogram to move one sprite toward the other.

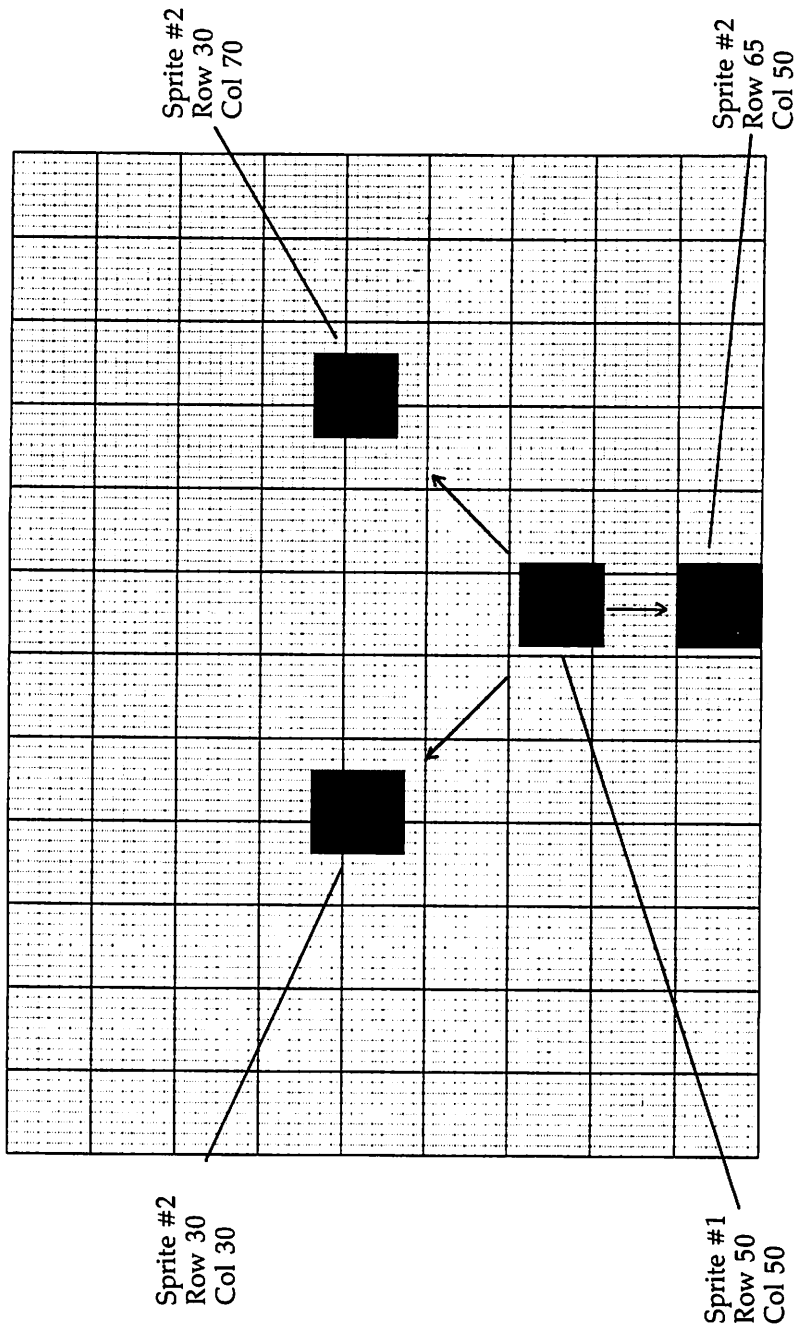
To understand how this works, first look at Figure 4-2. It shows the location for sprite #1 as dot row 50 and dot column 50. For the first example, assume that sprite #2 was located at dot row 30 and dot column 30. Subtracting the dot row for sprite #1 from the dot row for sprite #2 gives the row velocity. Subtracting the dot column for sprite #1 from the dot column for sprite #2 gives the column velocity. These numbers can then be used by the CALL MOTION subprogram. For example:

```
30 (dot row sprite#2)
-50 (dot row sprite#1)
-20 (row velocity)

30 (dot column sprite#2)
-50 (dot column sprite#1)
-20 (column velocity)
```

Recall from the section on CALL MOTION (Chapter 3) that a negative row velocity moves a sprite upward, while a negative column velocity moves a sprite to the left. In this case, a row velocity of -20 and column velocity of -20 would move sprite #1 up and to the left—directly toward sprite #2.

Figure 4-2. Relative Sprite Direction



Advanced Sprite Handling Techniques

In similar fashion, if sprite #2 was located at dot row 30 and dot column 70, then the above formula would produce a row velocity of -20 and a column velocity of 20 . These velocities would move sprite #1 up and to the right. If sprite #2 was located at dot row 65 and dot column 50, the formula would produce a row velocity of 15 and a column velocity of 0 . As a result, sprite #1 would move straight down.

The only problem with this approach is that a sprite's speed is determined by its distance from another sprite. But since distance would not generally be the controlling factor, this could present a problem. It is easily remedied, however, by dividing the resulting row and column velocities by their respective absolute values. Using the first example above, -20 divided by 20 (the absolute value of -20) results in -1 . A row velocity of -1 and column velocity of -1 still moves the sprite up and to the left. Since the values produced by the division will always be 1 or -1 , you could then multiply them by the speed factor desired (for example, $-1 * 8 = -8$).

The following program, "Sprite Chase," demonstrates how these concepts can actually be used. The program displays two sprites on the screen. The red sprite will attempt to chase the blue sprite, and the object is to use the E, S, D, and X (arrow) keys to move the blue sprite to evade the red one.

How Sprite Chase Works

Line(s)

| | |
|---------|---|
| 100 | Clear the screen. |
| 110 | Define the pattern used by the two sprites. |
| 120 | Display the program title on the screen. |
| 130-140 | Define the two sprites. Sprite #1 is the red "chase" sprite and is initially positioned at row 90, column 125. Sprite #2 is the blue "chased" sprite and is initially positioned at row 120, column 20. |
| 150-160 | Check for keyboard input. If a key has been pressed, then line 160 uses logical expressions to determine the values of X and Y. These values are then multiplied by 8 in order to give sprite #2 a relative speed of 8. |
| 170 | Determine the positions of both sprites. The row and column for sprite #1 are placed in R1 and C1. The row and column for sprite #2 are placed in R2 and C2. |
| 180-210 | These lines make sure that sprite #1 is not moved off the screen because of row or column wraparound. If |

Advanced Sprite Handling Techniques

the row position is less than 13 or greater than 170, vertical motion is inhibited by the MIN and MAX functions in lines 180-190. If the column position is less than 16 or greater than 232, horizontal motion is inhibited by MIN and MAX functions in lines 200-210. The overall effect is to prevent the sprite from moving off the screen.

- 220 Put sprite #2 in motion in the direction indicated by the X and Y values calculated in line 160. The idea is to keep it away from the "chasing" sprite (#1).
- 230-240 Set variables A and B to the values required by the CALL MOTION subprogram in line 270 to move sprite #1 toward sprite #2. The values are determined by subtracting the row and column position of sprite #1 from the row and column position of sprite #2. The resulting values are then divided by their respective absolute values in order to equalize the row and column velocities. Finally, the equalized values are multiplied by 9, which gives sprite #1 a relative speed of 9. Doing this gives sprite #1 a slight speed advantage. The IF statements in these two lines make sure that the subtraction does not cause a divide-by-zero error.
- 250 Check to see if the two sprites are at least six pixels apart.
- 260 If the two sprites are not six pixels apart, sprite #2 has been caught. This line displays the GOTCHA message and ends the program.
- 270 If the two sprites are more than five pixels apart, sprite #2 has not yet been caught. The CALL MOTION subprogram command in this line gets sprite #1 moving toward sprite #2 by using the values calculated in lines 230-240.
- 280 Increment the score counter and display it on the screen.
- 290 Branch back to start the process over.

Program 4-1. Sprite Chase

```
100 CALL CLEAR
110 CALL CHAR(96,"00183C7EFF7E3C1800")
120 DISPLAY AT(1,8):"SPRITE CHASE"
130 CALL SPRITE(#1,96,7,90,125)
140 CALL SPRITE(#2,96,5,120,20)
150 CALL KEY(3,K,S)
160 X=((K=83)-(K=68))*8 :: Y=((K=88)-(K=69))*8
170 CALL POSITION(#1,R1,C1,#2,R2,C2)
```

```
180 IF R2<13 THEN Y=MIN(Y,0)
190 IF R2>170 THEN Y=MAX(Y,0)
200 IF C2<16 THEN X=MAX(X,0)
210 IF C2>232 THEN X=MIN(X,0)
220 CALL MOTION(#2,-Y,X)
230 IF R1=R2 THEN A=0 ELSE A=(R2-R1)/ABS(R2-R1)*9
240 IF C1=C2 THEN B=0 ELSE B=(C2-C1)/ABS(C2-C1)*9
250 IF ABS(R1-R2)>5 OR ABS(C1-C2)>5 THEN 270
260 CALL DELSPRITE(ALL):: DISPLAY AT(10,12): "GOTCH
  A!" :: STOP
270 CALL MOTION(#1,A,B)
280 CT=CT+1 :: DISPLAY AT(24,10): "SCORE:";CT
290 GOTO 150
```

Determining When Sprites Are Coincident

In programs involving sprites, it is often important to know if two sprites are in the location (*coincident*) or if a sprite is at a specified screen location. One way to do this is to get the location of each sprite by using CALL POSITION and then comparing the locations. This is the method used in the previous program.

Checking the values from a CALL POSITION subprogram isn't really necessary, however, since Extended BASIC provides a subprogram specifically designed to detect sprite coincidence. This subprogram is CALL COINC, which may have any of the following formats:

Format 1

CALL COINC(#sprite number, #sprite number, tolerance,
numeric variable)

Format 2

CALL COINC(#sprite number, dot row, dot-column,
tolerance, numeric variable)

Format 3

CALL COINC(ALL, numeric variable)

When Format 1 is used, the sprite identified by the first sprite number is checked for coincidence against the sprite identified by the second sprite number. Both numbers must be integers in the range 1-28 and may be numeric literals, numeric variables, or numeric expressions. Coincidence is determined by the top left pixel of each sprite. When these two pixels occupy the same row and column location on the screen, the sprites are considered coincident and a value of

Advanced Sprite Handling Techniques

-1 is placed in *numeric variable*. When the two sprites are not coincident, numeric variable contains a value of 0.

Tolerance is used to expand the area which two sprites can occupy and still be considered coincident. For example, if the tolerance is 0, then the top left pixel of each sprite must be at exactly the same location to be considered coincident. However, if a tolerance of 3 is specified, then the sprites are considered coincident as long as the left-hand corners of the two sprites are within three pixels of each other. Such tolerances are especially useful when moving sprites make it difficult to detect exact coincidence.

When Format 2 is used, the sprite identified by sprite number is checked for coincidence against the screen location identified by dot-row and dot-column. Coincidence exists when the top left corner of the sprite is at the specified location or within the range indicated by tolerance.

With Format 3, coincidence is reported in numeric variable whenever *any* pixels in two or more sprites occupy the same location.

The following short program demonstrates how CALL COINC works. It displays two sprites on the screen and lets you move them with the E, S, D, and X keys. The display at the bottom of the screen indicates whether or not the sprites are coincident.

```
100 CALL CLEAR
110 CALL CHAR(96,"FFFFFFFFFFFFFFFF")
120 CALL SPRITE(#1,96,2,70,125)
130 CALL SPRITE(#2,96,16,90,125)
140 CALL KEY(3,K,S)
150 X=(K=83)-(K=68) :: Y=(K=88)-(K=69)
160 CALL MOTION(#1,-Y,X)
170 CALL COINC(#1,#2,0,A)
180 IF A= -1 THEN DISPLAY AT (23,10):"COINCIDENT"
    ELSE DISPLAY AT (23,10):" "
190 GOTO 140
```

Notice that the tolerance of the CALL COINC in line 170 is initially set to 0. The COINCIDENT message will appear only when the two sprites are in exactly the same location. If you change the tolerance to 3 and RUN the program again, you will see the COINCIDENT message displayed as long as the top left corners of the two sprites are within three pixels of each other.

The next program, "Air Defense," demonstrates how CALL COINC can actually be used in a program. The object of the game is to shoot down the alien spacecraft. Your anti-aircraft gun is aimed by pressing the X and D keys and fired by pressing the space bar.

How Air Defense Works

| Line(s) | |
|---------|--|
| 100 | Clear the screen. |
| 110 | Seed the random number generator. |
| 120 | Initialize program variables. |
| 130-220 | Define the patterns used in the program. ASCII codes 96-98 define the pattern for a gun pointing straight up; code 98 defines the pattern for a gun pointing up and to the right. ASCII codes 100-102 define the patterns used for various alien ships. ASCII code 104 defines the pattern used for the base of the screen. ASCII codes 112 and 113 define patterns that simulate an exploding ship. |
| 230-250 | Draw the ground on the screen. |
| 260 | Create the antiaircraft gun sprite on the screen. It initially points straight up. |
| 270 | Create an alien ship on the screen. The ship's pattern is randomly chosen from one of the three available. The ship moves from right to left, at random speed, somewhere between rows 20 and 145. |
| 280 | Check to see if a key was pressed. |
| 290-300 | Calculate the player's rating and display it on the screen. |
| 310 | If no key was pressed on the console, go back and check again. |
| 320-360 | Determine which key was pressed. If it was S, the gun is pointed to the left; if it was D, the gun is pointed to the right. In either case, the new pattern for the gun is assigned to sprite #1 in line 350, and the program goes back to check for another key. If the space bar was pressed, go to the routine to fire the gun. |
| 370-400 | Fire the gun. Line 370 creates the projectile pattern on the screen. Its initial location is determined by the direction in which the gun is pointing. Line 380 adds to the SHOT accumulator. Line 390 sets the row velocity to -30 and the column velocity to 30,0, or -30, depending on which way the gun is pointed. Line 400 puts the projectile in motion. |
| 410 | Check to see if the projectile has reached the edge of the screen. If so, the sprite is deleted. |

Advanced Sprite Handling Techniques

- 420-430 Check to see if the ship was hit by the projectile. If the projectile sprite (#2) and the ship sprite (#3) are coincident within a tolerance of 5, the ship was hit. If the ship wasn't hit, go back and check again until the projectile is off the screen.
- 440-520 Register a hit. Line 440 changes the ship's color to red. Line 450 deletes the projectile sprite. Lines 460 and 480 change the ship's pattern to the explosion patterns. Lines 470 and 490 are timing loops between the pattern changes. Line 500 deletes the ship sprite. Line 510 counts the hit. Line 520 goes back and creates another alien ship to start the whole thing over.

Program 4-2. Air Defense

```
100 CALL CLEAR
110 RANDOMIZE
120 PAT=97 :: SHOT=.1
130 CALL CHAR(96,"0000C06030187EFF")
140 CALL CHAR(97,"0000181818187EFF")
150 CALL CHAR(98,"000003060C187EFF")
160 CALL CHAR(100,"0000003C7EFF7E3C")
170 CALL CHAR(101,"187EFFFF24428100")
180 CALL CHAR(102,"00007E7EFF7E7E00")
190 CALL CHAR(103,"00000000000181800")
200 CALL CHAR(104,"FFFFFFFFFFFFFFFF"):: CALL COLOR
    (10,13,1)
210 CALL CHAR(112,"0100001018004081")
220 CALL CHAR(113,"1080810082828114")
230 FOR L=22 TO 24
240 CALL HCHAR(L,1,104,32)
250 NEXT L
260 CALL SPRITE(#1,97,2,162,125)
270 CALL SPRITE(#3,100+INT(RND*3),5,20+INT(RND*125),
    255,0,-(10+INT(RND*12)))
280 CALL KEY(3,K,S)
290 RATE=HIT/SHOT*100
300 DISPLAY AT(1,8):USING "##### ###":"RATING:",
    RATE
310 IF S=0 THEN 280
320 IF K=83 THEN PAT=MAX(PAT-1,96)
330 IF K=68 THEN PAT=MIN(PAT+1,98)
340 IF K=32 THEN 370
350 CALL PATTERN(#1,PAT)
360 GOTO 280
370 CALL SPRITE(#2,103,7,156,125-((97-PAT)*6))
380 SHOT=SHOT+1
```

Advanced Sprite Handling Techniques

```
390 R=-30 :: C=((PAT-97)*30)
400 CALL MOTION(#2,R,C)
410 CALL POSITION(#2,X,Y):: IF X<12 OR Y<5 OR Y>25
    0 THEN CALL DELSPRITE(#2):: GOTO 280
420 CALL COINC(#2,#3,5,CO)
430 IF CO=0 THEN 410
440 CALL COLOR(#3,9)
450 CALL DELSPRITE(#2)
460 CALL PATTERN(#3,112)
470 FOR L=1 TO 30 :: NEXT L
480 CALL PATTERN(#3,113)
490 FOR L=1 TO 30 :: NEXT L
500 CALL DELSPRITE(#3)
510 HIT=HIT+1
520 GOTO 270
```

Determining Sprite Distances

The CALL DISTANCE subprogram command is used to determine the distance between two sprites or between one sprite and a screen location. It may have either of two formats:

Format 1

CALL DISTANCE(#sprite number,#sprite number,numeric variable)

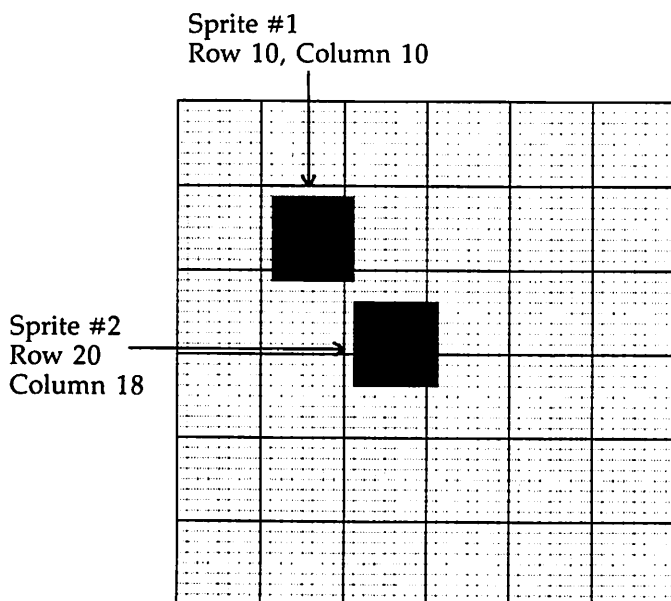
Format 2

CALL DISTANCE(#sprite number,dot row,dot column,numeric variable)

When Format 1 is used, the distance between the sprite identified by the first sprite number and the sprite identified by the second sprite number is found and placed in numeric variable. The location of both sprites is determined by their top left pixel. Both sprite numbers must be integers between 1-28 and may be numeric literals, numeric variables, or numeric expressions.

The value placed in numeric variable is determined by the following computation: The difference between the dot row of the first sprite number and the dot row of the second sprite number is found and squared. The difference between the dot-column of the first sprite number and dot column of the second sprite number is found and squared. The two squares are then added together and placed in numeric variable. If this sum is greater than 32,767, then 32,767 is placed in numeric variable. The actual distance between the two sprites is the square root of numeric variable.

Figure 4-3. Distance Between Sprites



$$\begin{array}{rcl}
 20 & \text{sprite \#2 row} & \\
 -10 & \text{sprite \#1 row} & \\
 \hline
 10 & & \\
 \\
 18 & \text{sprite \#2 column} & \\
 -10 & \text{sprite \#1 column} & \\
 \hline
 8 & & \\
 \\
 10^2 = & 100 & \\
 + 8^2 = & 64 & \\
 \hline
 164 & \text{distance} &
 \end{array}$$

Figure 4-3 shows two sprites. The first is located at dot row 10 and dot column 10. The second is located at dot row 20 and dot column 18. The difference between the dot rows (10) is squared, resulting in 100. The difference between the dot columns (8) is squared, resulting in 64. The sum of the two squares (164) is placed in numeric variable.

When Format 2 is used, the distance between the sprite identified by sprite number and the location identified by dot row and dot column is found and placed in numeric variable. The computation used to find the distance is the same as the one for two sprites.

Advanced Sprite Handling Techniques

The following short program displays the values computed for numeric variable. It places two sprites on the screen and lets you move one of them with the E, S, D, and X keys. The distance between the sprites is continuously displayed at the bottom of the screen.

```
100 CALL CLEAR
110 CALL CHAR(96,"FFFFFFFFFFFFFFFF")
120 CALL SPRITE(#1,96,2,70,125)
130 CALL SPRITE(#2,96,16,90,125)
140 CALL KEY(3,K,S)
150 X=(K=83)-(K=68) :: Y=(K=88)-(K=69)
160 CALL MOTION(#1,-Y,X)
170 CALL DISTANCE (#1,#2,D)
180 DISPLAY AT(23,10):"DISTANCE";D
190 GOTO 140
```

CALL DISTANCE may be used in place of CALL COINC in your programs. While CALL COINC is used to test for coincidence, CALL DISTANCE has the additional benefit of providing the actual distance between sprites. This allows your program to determine *how close* as well as *hit* or *miss*.

The next program, "Meteors," demonstrates how CALL DISTANCE can be used in a program in place of CALL COINC. The program displays a starship in the middle of the screen. The ship can be pointed up, down, left, or right by using the E, S, D, and X keys. The object of the game is to shoot the meteors by pressing the space bar. You can fire only one shot at a time. If your ship is struck by one of the meteors, the ship is destroyed. You have three ships to lose before the game is over. Stay alert!

How Meteors Works

Line(s)

| | |
|---------|--|
| 100 | Clear the screen. |
| 110 | Seed the random number generator. |
| 120-200 | Define the patterns used in the program. ASCII code 96 defines the ship pointing up; code 97 defines the ship pointing right; code 98 defines the ship pointing down; code 99 defines the ship pointing left. ASCII codes 104-106 are used to define three patterns for the meteors. ASCII code 112 defines the projectile fired by the ship. ASCII code 120 defines an "explosion" character. |

Advanced Sprite Handling Techniques

- 210-230 Set the screen to dark blue, initialize the ship counter, and display the ship counter and score on the screen.
- 240 Place the ship sprite (#1) at the center of the screen, pointing up (ASCII code 96).
- 250 Initialize the row and column velocities for the projectile so that it corresponds to the direction in which the ship is pointing. These velocities will change as the direction of the ship changes.
- 260-280 Call the routine that generates the three meteor sprites.
- 290-340 Check for keyboard input. If the space bar was pressed, go to the routine to fire the projectile. If the E, S, D, or X key is pressed, change the ship's pattern to the one corresponding to that direction and load the appropriate row and column velocities for the projectile.
- 350-370 After checking the keys, check to see if one of the three meteors has collided with the ship. The specified distance of 73 allows for a collision if the ship and a meteor are within roughly six pixels of each other. This routine is within the main program loop since a meteor could collide with the ship at any time.
- 380 After checking for keyboard input and collisions, go back and do it again.
- 390-480 This routine takes command when the space bar is pressed, indicating that a projectile has been fired. Line 390 fires the projectile based on the direction in which the ship is pointing (RV and CV). Line 400 begins a loop that lasts for the duration of the projectile's flight. Within this loop, lines 410-440 check to see if the projectile hit one of the meteors. Again, a distance of 73 was used. If the projectile did hit a meteor, program control is passed to line 610. After the loop in line 400 is completed, the projectile sprite (#2) is deleted. Otherwise, the projectile would continue to wrap around the screen. If there are still meteors on the screen, line 460 branches back to the keyboard check routine. If all the meteors have been destroyed, line 470 creates three more of them. One important thing to notice is the second CALL DISTANCE in line 420. This statement continues to check for ship and meteor collision even while the projectile is being tracked.
- 490-530 Subroutine to create three meteors. The row and column velocities are randomly generated.
- 540-600 Subroutine for a ship and meteor collision. Lines 540-550 briefly change the ship's pattern to the explosion character. The ship is then deleted in line 560. Line

570 subtracts 1 from the ship count. If there are still ships left, line 590 goes back to line 230 to place a new ship on the screen. If there are no more ships, the program is ended in line 600.

610-650 Subroutine for a meteor hit by a projectile. Line 610 changes the affected meteor to the explosion character. Line 620 updates the meteor counter and score. Line 630 deletes the meteor sprite. Lines 640-650 display the score and return to the calling routine.

Program 4-3. Meteors

```
100 CALL CLEAR
110 RANDOMIZE
120 CALL CHAR(96,"18183C3C3C7EFFFF")
130 CALL CHAR(97,"C0E0FCFFFFFCE0C0")
140 CALL CHAR(98,"FFFF7E3C3C3C1818")
150 CALL CHAR(99,"03073FFFFF3F0703")
160 CALL CHAR(104,"387E3F7EFFFF7F1F")
170 CALL CHAR(105,"3C7E7EFFFFF7E3C")
180 CALL CHAR(106,"7C7F3F3E1C3E7E3F")
190 CALL CHAR(112,"0000001818000000")
200 CALL CHAR(120,"01124201441811C3")
210 CALL SCREEN(5)
220 SHIPS=3
230 DISPLAY AT(24,6):"SHIPS:";SHIPS-1 :: DISPLAY A
    T(24,15):"SCORE:";SC
240 CALL SPRITE(#1,96,15,90,125)
250 RV=-20 :: CV=0
260 FOR SP=3 TO 5
270 GOSUB 490
280 NEXT SP
290 CALL KEY(3,K,S)
300 IF K=32 THEN 390
310 IF K=83 THEN CALL PATTERN(#1,99):: RV=0 :: CV=
    -20
320 IF K=68 THEN CALL PATTERN(#1,97):: RV=0 :: CV=
    20
330 IF K=69 THEN CALL PATTERN(#1,96):: RV=-20 :: C
    V=0
340 IF K=88 THEN CALL PATTERN(#1,98):: RV=20 :: CV
    =0
350 FOR L2=3 TO 5
360 CALL DISTANCE(#1,#L2,DI):: IF DI<73 THEN 540
370 NEXT L2
380 GOTO 290
390 CALL SPRITE(#2,112,16,90,125,RV,CV)
400 FOR L=1 TO 6
```

Advanced Sprite Handling Techniques

```
410 FOR SP=3 TO 5
420 CALL DISTANCE(#2,#SP,DIST):: CALL DISTANCE(#1,
    #SP,DI):: IF DI<73 THEN 540
430 IF DIST<73 THEN GOSUB 610
440 NEXT SP :: NEXT L
450 CALL DELSPRITE(#2)
460 IF CT<3 THEN 290
470 FOR SP=3 TO 5 :: GOSUB 490 :: NEXT SP
480 GOTO 290
490 CALL SPRITE(#SP,104+INT(RND*3),2,240,90)
500 SRV=3+INT(RND*5):: IF RND<.5 THEN SRV=-SRV
510 SCV=3+INT(RND*5):: IF RND<.5 THEN SCV=-SCV
520 CALL MOTION(#SP,SRV,SCV)
530 CT=0 :: RETURN
540 CALL PATTERN(#1,120)
550 FOR L2=1 TO 15 :: NEXT L2
560 CALL DELSPRITE(ALL)
570 SHIPS=SHIPS-1
580 IF SHIPS=0 THEN 600
590 FOR L=1 TO 300 :: NEXT L :: GOTC 230
600 STOP
610 CALL PATTERN(#SP,120)
620 CT=CT+1 :: SC=SC+1
630 CALL DELSPRITE(#2,#SP)
640 DISPLAY AT(24,21):SC
650 L=99 :: SP=9 :: RETURN
```

Factors Affecting POSITION, COINC, and DISTANCE

There are two factors you should keep in mind when using CALL POSITION, CALL COINC, and CALL DISTANCE in your programs. Both have to do with speed.

The first involves the row and column velocities of the sprites used in the program. If these velocities are high, the values returned from the subprograms may not be right. The reason is simple: The three subprograms get the values at the instant they are executed in the program, but when sprites are moving very fast, the subprograms have a much smaller margin of error, or window, to work with. The results, of course, are missed coincidences and incorrect distances.

The second problem is similar. It has to do with the relatively slow speed of BASIC. As an example, consider a program that contains eight sprites. You want the program to check for coincidence between sprite #1 and the other seven sprites. If your program contains seven CALL COINC subprogram statements in a row, the chances are very good that

the final two or three will miss the coincidence. By the time the program gets to those lines, the sprites have moved out of range.

There are ways to overcome, or at least reduce, these problems. Use efficient programming techniques (multiple statement lines, short variable names, etc.). Check your program for needless GOTOs or dead code. Where possible, combine repetitive code into a single subroutine. Make sure your program is well-designed. Do you need to check every sprite for coincidence in the same place? Can you use the ALL keyword instead of checking each sprite? The list goes on.

Writing a Graphics Program

By this point, you should be familiar with defining characters, displaying characters on the screen, assigning character colors, defining sprites, controlling sprite motion, and determining sprite status. But understanding these techniques is only the first step toward working up a graphics program of your own.

This section will show you how to incorporate such techniques into a game program of your own. It will explain, step by step, how a graphics program evolves from a rough idea in the programmer's mind into the finished product.

The Scenario

The program will be a one-player maze game called "Mouse Maze." The object of the game will be to get the mouse from the top of the maze to the mouse hole at the bottom. It might be easy—but unfortunately for the mouse, the maze will also be occupied by two hungry cats.

The cats will have the ability to chase the mouse. They'll have enough intelligence to know where the mouse is, but not enough to backtrack when they reach barriers in the maze. Neither the cats nor the mouse can jump the maze barriers. Both the cats and the mouse will move at the same speed. The mouse's only chance for survival, therefore, is to outsmart the cats. The player will control the mouse by using the E, S, D, and X keys.

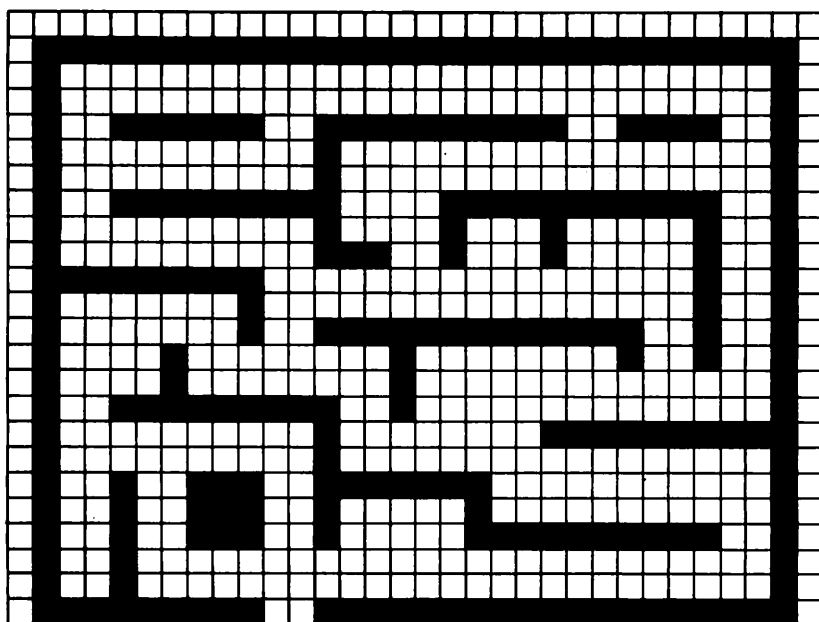
The complete program listing is presented at the end of this chapter. You should refer to it when particular sections of code are being explained.

Defining the Graphics

Once the scenario is set, the next step is to identify and define the required graphics characters. From the game description you can see that two types of graphics are necessary for this game.

Obviously, there must be a maze. Since the maze is, in effect, the playing board, it should be designed first. The best way to design a maze—or, for that matter, any other fixed screen pattern—is to draw it on graph paper marked for 32 columns by 24 lines. The maze used in the program is shown in Figure 4-4.

Figure 4-4. Mouse Maze Screen Layout



After plotting the maze, you must decide on a character to make up the maze walls. You could use a standard ASCII pattern such as the X, but that would be a waste of the flexibility that your TI provides.

It would be more effective visually to use a custom character, and that is what has been done here. Line 120

defines ASCII code 40 as a square with a dark red border and a gray interior. When this character is used to make the maze barrier, it gives the impression of bricks or blocks.

There still remains the matter of displaying your maze on the screen. This is accomplished easily enough by looking at the maze design, finding the starting location and length of the barrier sections, and using CALL HCHAR and CALL VCHAR to place them on the screen. Lines 370–590 do just that.

But there's more to this maze game than its maze. The sprites must also be defined. This particular scenario involves two basic sprite shapes, one for the cats and one for the mouse, and they should be displayed with reasonably high resolution in order to provide realism. Double-sized unmagnified sprites (line 240) would be the best choice, since they provide a total of 256 pixels per animal.

Note that the size of the sprites determines the minimum width of the maze corridors. In this case, with double-sized unmagnified sprites, the corridors must be at least two characters wide in every dimension. Remember, double-sized sprites measure two characters by two characters.

Though you only have two types of creatures loose in your maze, remember that each one can move in any of four different directions. Consequently, *four* distinct sprites will be required for each. Figure 4-5 shows the four patterns for the mouse, which are defined in the program as four-character ASCII codes 96, 100, 104, and 108, in lines 130–160. Figure 4-6 depicts the patterns for the cats, which are defined as four-character ASCII codes 112, 116, 120, and 124, in lines 170–200.

The mouse is placed on the screen, in line 330, at row 17, column 161. The cats, on the other hand, are initially located at two of six possible locations (lines 340–350), which makes the game less repetitious. The six possible locations are contained in the DATA statements in lines 1200 and 1210. These are READ in lines 280–300. The two locations actually used are randomly selected in lines 250–260. Line 270 makes sure that the two locations are different.

Finally, the game should include some sort of animation depicting the mouse being caught by a cat. This is accomplished by alternating the three patterns shown in Figure 4-7. They are defined in lines 210–230 as four-character ASCII codes 128, 132, and 136.

Figure 4-5. The Mouse

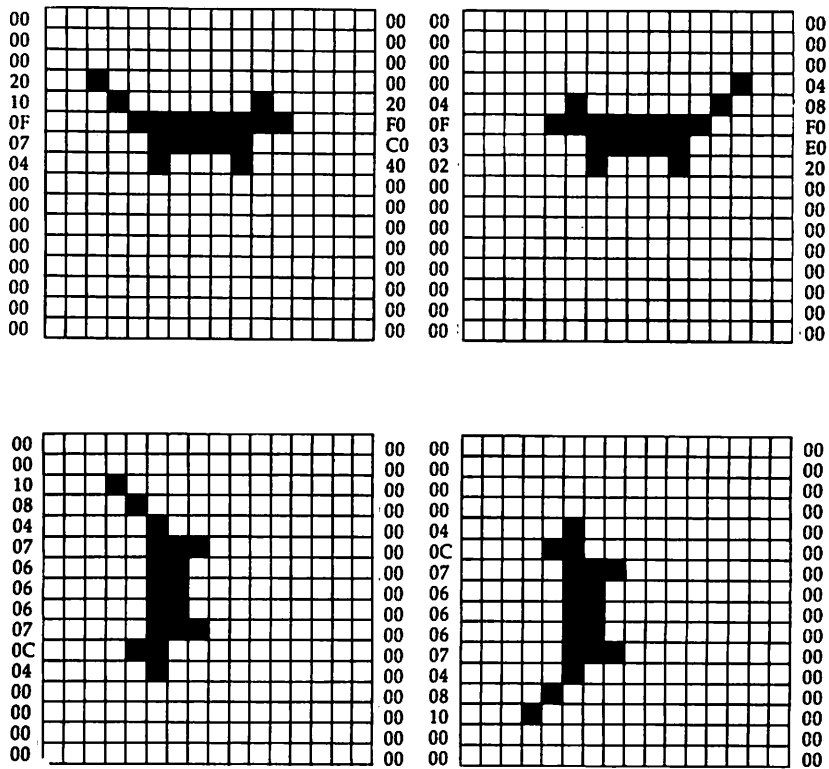
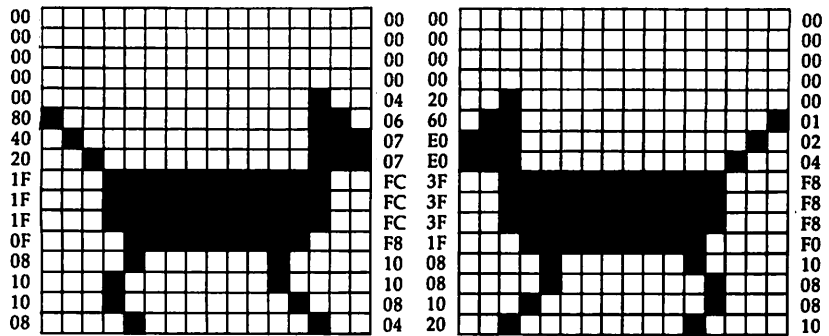


Figure 4-6. The Cats



Advanced Sprite Handling Techniques

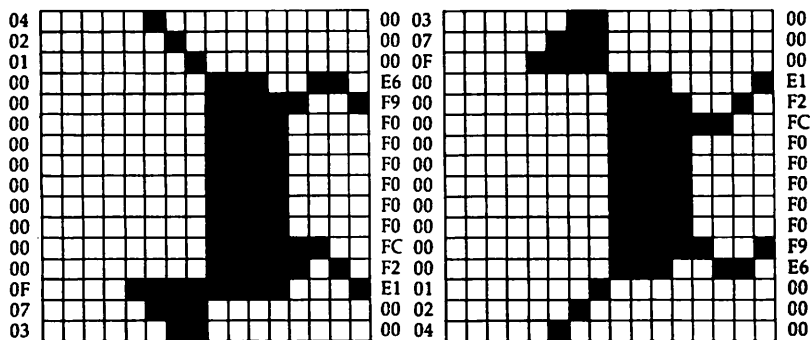
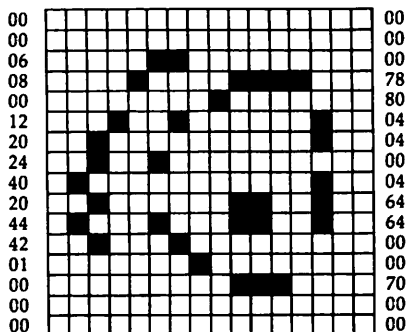
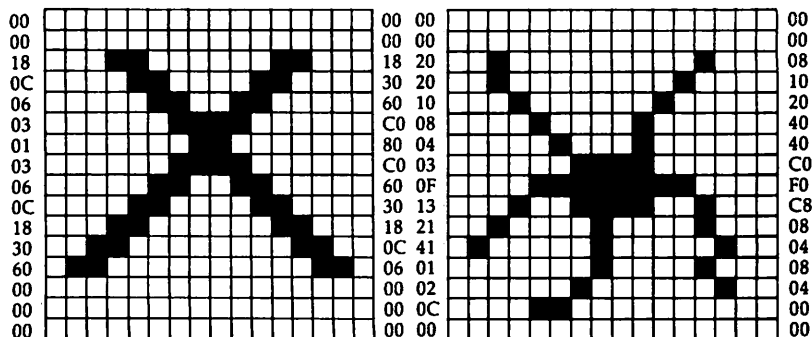


Figure 4-7. Caught



Controlling the Action

Since this game involves ongoing action, a main control loop must be established. This loop must determine if the mouse has been caught. It must also move the cats toward the mouse without letting them jump the barrier, and it has to see if the E, S, D, or X key is pressed and then move the mouse accordingly. Finally, it has to determine if the mouse has safely reached the mouse hole.

The control loop begins in line 600. The first thing it does is get the position of the mouse and the two cats. If the mouse is past row 172, it has reached safety. The program then branches to line 1140, which ends the game. Otherwise, the program checks COINC in lines 610 and 620 to see if the mouse has been caught. If the mouse is within eight pixels of either cat, the mouse is caught. In that case, the program branches to 1060 and initiates a loop that alternates ASCII codes 128, 132, and 136. The effect is a cartoon-style cat and mouse fight.

If coincidence is not reported, the program calls a routine to move the cats (lines 630–680). The values from the earlier CALL POSITION are used to determine the row and column velocities necessary to move the cat toward the mouse (line 710). If the mouse is further away vertically than horizontally (line 720), the cat attempts to move vertically (lines 790–840). Otherwise, the cat attempts to move horizontally (lines 730–780).

If the cat attempts to move vertically, but is blocked by the barrier (line 820), it then attempts to move horizontally. If it is still blocked, no movement occurs. The same applies when the first attempt is horizontal (line 760).

Since the sprite measures two characters by two characters, a cat can move vertically only when both positions in the vertical direction are blank. To check for this condition, the program converts the sprite row (adjusted for the move) and column from the CALL POSITION into character row and column (lines 790–800). It then issues two CALL GCHARs to determine if the new locations are blank. If they are, the move is made (line 670), and the cat pattern for that direction is used (line 830). If the new locations aren't blank, and no horizontal movement attempt has been made (IF SW=0 in line 820), the program branches to the horizontal movement routine (GOTO 730 in line 820). Horizontal movement is controlled in similar fashion in lines 750 and 760.

After the cats have been moved, the program checks for keyboard input (line 690). If there is keyboard input, lines 850–880 determine which key and invoke the appropriate routine. For example, if the S key was pressed (moving the mouse to the left), the program branches to line 900. The new location for the mouse is determined (line 900), and a CALL GCHAR is issued (line 910) to see what's there. If the new location is blank, the mouse is moved and its pattern changed to coincide with that direction (line 920). If the new location isn't blank, no movement is made. In either case, the program RETURNS (line 930) and starts the process all over.

Program 4-4. Mouse Maze

```
100 CALL CLEAR
110 RANDOMIZE
120 CALL CHAR(40,"FF8181818181FF"):: CALL COLOR(
    2,7,15)
130 CALL CHAR(96,"000000000040F030200000000000000000
    00000000408F0E02000000000000000000")
140 CALL CHAR(100,"00000020100F07040000000000000000
    00000000020F0C04000000000000000000")
150 CALL CHAR(104,"000010080407060606070C0400000000
    00000000000000000000000000000000")
160 CALL CHAR(108,"00000000040C07060606070408100000
    00000000000000000000000000000000")
170 CALL CHAR(112,"000000002060E0E03F3F3F1F0808102
    000000000000010204F8F8F8F010080810")
180 CALL CHAR(116,"00000000008040201F1F1F0F0810100
    80000000004060707FCFCFCF810100804")
190 CALL CHAR(120,"040201000000000000000000000000F070
    30000000E6F9F0F0F0F0F0F0FCF2E10000")
200 CALL CHAR(124,"03070F000000000000000000000001020
    40000E1F2FCF0F0F0F0F0F0F0F9E6000000")
210 CALL CHAR(128,"0000006030180C06030103060C18000
    0000000060C183060C080C06030180000")
220 CALL CHAR(132,"0000060800112044402044010000000
    000003C80040404040460640000380000")
230 CALL CHAR(136,"00004040201008030F1121418202040
    000000810204040CF0E1080408040101")
240 CALL MAGNIFY(3)
250 LC1=1+INT(RND*6)
260 LC2=1+INT(RND*6)
270 IF LC1=LC2 THEN 260
280 FOR L=1 TO 6
290 READ CRL(L),CCL(L),CCH(L)
300 NEXT L
```

Advanced Sprite Handling Techniques

```
310 X(1)=CRL(LC1):: X(2)=CRL(LC2)
320 Y(1)=CCL(LC1):: Y(2)=CCL(LC2)
330 CALL SPRITE(#1,100,16,17,161)
340 CALL SPRITE(#2,CCH(LC1),2,CRL(LC1),CCL(LC1))
350 CALL SPRITE(#3,CCH(LC2),2,CRL(LC2),CCL(LC2))
360 CALL SCREEN(6)
370 CALL HCHAR(2,2,40,30)
380 CALL HCHAR(5,5,40,6):: CALL HCHAR(5,13,40,10):
: CALL HCHAR(5,25,40,4)
390 CALL HCHAR(8,5,40,8):: CALL HCHAR(8,18,40,11)
400 CALL HCHAR(11,3,40,8)
410 CALL HCHAR(13,13,40,4):: CALL HCHAR(13,17,40,9
)
420 CALL HCHAR(16,5,40,9)
430 CALL HCHAR(17,22,40,9)
440 CALL HCHAR(19,8,40,3):: CALL HCHAR(19,13,40,7)
450 CALL HCHAR(20,8,40,3)
460 CALL HCHAR(21,8,40,3):: CALL HCHAR(21,19,40,10
)
470 CALL HCHAR(24,2,40,9):: CALL HCHAR(24,13,40,19
)
480 CALL VCHAR(2,2,40,23)
490 CALL VCHAR(19,5,40,5)
500 CALL VCHAR(14,7,40,2)
510 CALL VCHAR(11,10,40,3)
520 CALL VCHAR(5,13,40,6):: CALL VCHAR(16,13,40,6)
530 CALL VCHAR(13,16,40,4)
540 CALL VCHAR(8,18,40,3)
550 CALL VCHAR(19,19,40,3)
560 CALL VCHAR(8,22,40,3)
570 CALL VCHAR(13,25,40,2)
580 CALL VCHAR(8,28,40,7)
590 CALL VCHAR(2,31,40,23)
600 CALL POSITION(#1,R1,C1,#2,R2(1),C2(1),#3,R2(2)
,C2(2)):: IF R1>172 THEN 1140
610 CALL COINC(#1,#2,8,CO):: IF CO<>0 THEN CO=2 ::
GOTO 1060
620 CALL COINC(#1,#3,8,CO):: IF CO<>0 THEN CO=3 ::
GOTO 1060
630 FOR L=1 TO 2
640 GOSUB 710
650 NEXT L
660 FOR L=2 TO 3
670 CALL LOCATE(#L,R2(L-1),C2(L-1))
680 NEXT L
690 CALL KEY(3,K,S):: IF S<>0 THEN GOSUB 850
700 GOTO 600
710 MS=0 :: SW=0 :: RV=R1-R2(L):: CV=C1-C2(L)
720 IF ABS(RV)>ABS(CV) THEN 790
```

Advanced Sprite Handling Techniques

```
730 IF CV<0 THEN COL=MIN(INT((C2(L)-8)/8)+1,32)ELS
    E COL=MIN(INT((C2(L)+16)/8)+1,32)
740 ROW=INT(R2(L)/8)+1
750 CALL GCHAR(ROW,COL,CH):: CALL GCHAR(ROW+1,COL,
    CH2)
760 IF CH<>32 OR CH2<>32 THEN IF SW=0 THEN SW=1 ::
    GOTO 790 ELSE RETURN
770 IF CV<0 THEN C2(L)=C2(L)-8 :: CALL PATTERN(#L+
    1,112)ELSE C2(L)=C2(L)+8 :: CALL PATTERN(#L+1,
    116)
780 RETURN
790 IF RV<0 THEN ROW=MIN(INT((R2(L)-8)/8)+1,24)ELS
    E ROW=MIN(INT((R2(L)+16)/8)+1,24)
800 COL=INT(C2(L)/8)+1
810 CALL GCHAR(ROW,COL,CH):: CALL GCHAR(ROW,COL+1,
    CH2)
820 IF CH<>32 OR CH2<>32 THEN IF SW=0 THEN SW=1 ::
    GOTO 730 ELSE RETURN
830 IF RV<0 THEN R2(L)=R2(L)-8 :: CALL PATTERN(#L+
    1,124)ELSE R2(L)=R2(L)+8 :: CALL PATTERN(#L+1,
    120)
840 RETURN
850 IF K=83 THEN 900
860 IF K=68 THEN 940
870 IF K=69 THEN 980
880 IF K=88 THEN 1020
890 RETURN
900 ROW=INT(R1/8)+1 :: COL=INT((C1-8)/8)+1
910 CALL GCHAR(ROW,COL,CH)
920 IF CH=32 THEN C1=C1-8 :: CALL PATTERN(#1,96)::
    CALL LOCATE(#1,R1,C1)
930 RETURN
940 ROW=INT(R1/8)+1 :: COL=INT((C1+16)/8)+1
950 CALL GCHAR(ROW,COL,CH)
960 IF CH=32 THEN C1=C1+8 :: CALL PATTERN(#1,100):
    : CALL LOCATE(#1,R1,C1)
970 RETURN
980 COL=INT(C1/8)+1 :: ROW=INT((R1-8)/8)+1
990 CALL GCHAR(ROW,COL,CH)
1000 IF CH=32 THEN R1=R1-8 :: CALL PATTERN(#1,108)
    :: CALL LOCATE(#1,R1,C1)
1010 RETURN
1020 COL=INT(C1/8)+1 :: ROW=INT((R1+16)/8)+1
1030 CALL GCHAR(ROW,COL,CH)
1040 IF CH=32 THEN R1=R1+8 :: CALL PATTERN(#1,104)
    :: CALL LOCATE(#1,R1,C1)
1050 RETURN
1060 FOR LP1=1 TO 5 :: FOR LP2=1 TO 9 STEP 4
```

Advanced Sprite Handling Techniques

```
1070 CALL PATTERN(#1,127+LP2):: CALL PATTERN(#CO,1
    37-LP2):: CALL COLOR(#1,9+INT(RND*4),#CO,7+IN
    T(RND*6))
1080 FOR LP3=1 TO 15 :: NEXT LP3
1090 NEXT LP2 :: NEXT LP1
1100 CALL DELSPRITE(#1):: CALL COLOR(#CO,2):: CALL
    PATTERN(#CO,116)
1110 DISPLAY AT(1,5):"PLAY AGAIN? Y OR N"
1120 CALL KEY(3,K,S):: IF S=0 THEN 1120
1130 IF K=89 THEN RESTORE :: GOTO 100 ELSE CALL DE
    LSPRITE(ALL):: CALL CLEAR :: STOP
1140 CALL DELSPRITE(ALL):: CALL CLEAR
1150 FOR LP1=1 TO 10
1160 DISPLAY AT(12,10):"HOORAY!" :: FOR LP2=1 TO 4
    0 :: NEXT LP2
1170 DISPLAY AT(12,1):" " :: FOR LP2=1 TO 40 :: NE
    XT LP2
1180 NEXT LP1
1190 STOP
1200 DATA 169,17,124,169,113,116,89,25,116
1210 DATA 81,185,112,41,113,112,105,81,124
```

5

Sound

5

Sound

As you've worked with your TI through the last few chapters, you've discovered that it has some remarkable graphics capabilities. But graphics is only part of the story. Your TI computer has an additional capability that can make things even more exciting: sound.

What Is Sound?

Sound is the result of changes in air pressure, which stimulate the ear drum and auditory nerves to produce the sensation of hearing. Such pressure changes radiate from a source—for instance, a vibrating guitar string—and travel through the air much like waves travel across a pond when you toss in a stone.

Sound can be characterized by volume, frequency, and vibrational pattern. Volume is simply the magnitude of the pressure waves, while frequency is nothing more than the number of pressure changes per second. Frequency is measured in hertz (Hz), or cycles per second, and most people can hear sounds ranging from about 20 to more than 20,000 Hz. The vibrational pattern embraces both the waveform (literally, the shape of the pressure waves) and the regularity of the pattern and determines a sound's tone color. For example, regularly varying waves will produce sounds having a definite pitch, while randomly varying waves will be heard as some form of noise.

Making Sound on the TI

Using the CALL SOUND subprogram, your TI can produce both musical tones and noise. CALL SOUND has the following format:

CALL SOUND(duration,frequency 1,volume 1,...,frequency 4,volume 4)

Duration is an integer in the range from 1 to 4250 or -1 to -4250 and can be a numeric literal, numeric variable, or numeric expression. It indicates the length of time in milliseconds (1000 milliseconds equals one second) that a tone or noise will last. Positive durations mean that a new CALL SOUND subprogram will not be executed until the previous

CALL SOUND has finished. If you use the negative sign, it tells the computer to execute a CALL SOUND immediately, regardless of what else is going on, and can be particularly useful in games or tutorials where sounds must change quickly. You can specify only one duration parameter per CALL SOUND subprogram.

The frequency parameter determines whether a sound is a musical tone or noise. For musical tones, it is an integer from 110 to 44733 that corresponds to the desired frequency. For noise, it is an integer from -1 to -8, depending on the type of noise that you want. Several different noise types are described in Table 5-1, along with their corresponding frequency values. In either case, the value can be a numeric literal, numeric variable, or numeric expression.

Table 5-1. Noises

| Frequency | Description |
|-----------|---|
| -1 | Periodic Noise Type 1 |
| -2 | Periodic Noise Type 2 |
| -3 | Periodic Noise Type 3 |
| -4 | Periodic Noise that varies with the frequency of the third tone specified |
| -5 | White Noise Type 1 |
| -6 | White Noise Type 2 |
| -7 | White Noise Type 3 |
| -8 | White Noise that varies with the frequency of the third tone specified |

Volume is an integer ranging from 0 to 30 and can also be a numeric literal, numeric variable, or numeric expression. Contrary to what you might expect, 0 represents the highest volume. Higher numbers produce lower volumes, with 30 being used for the softest possible sounds.

A maximum of three frequencies and one noise form, each with its own volume parameter, may be specified in the same CALL SOUND statement. All of the sounds will then be produced simultaneously.

The best way to learn your TI's sound capabilities is to experiment. You can RUN the following examples to get started, but don't hesitate to try different values and create some sounds of your own.

Frequency vs. Pitch

```
100 FOR F=110 TO 1760 STEP 50
110 CALL SOUND (500,F,0)
120 NEXT F
```

Duration

```
100 FOR D=100 to 4250 STEP 1000
110 CALL SOUND (D, 262,0)
120 FOR X=1 TO 1500 :: NEXT X
130 NEXT D
```

Volume

```
100 FOR V=0 TO 30
110 CALL SOUND (500,262,V)
120 FOR X=1 TO 500 :: NEXT X
130 NEXT V
```

Noise

```
100 FOR N=1 TO -8 STEP -1
110 CALL SOUND (3000,N,0)
120 FOR X=1 TO 1500 :: NEXT X
130 NEXT N
```

Harmony

```
100 CALL SOUND (3000,262,0,330,0,392,0)
```

Discord

```
100 CALL SOUND(3000,110,0,672,0,151,0)
```

Turning Sound into Music

As you've probably discovered, your TI can produce some interesting sounds. By themselves, those sounds can certainly provide an evening's entertainment or add new dimensions to your games. Your computer can also be a musician, and this section will show you how to tap its talents.

The following section is a short introduction to musical notation. If you already know how to read music, you might want to skip ahead. If not, this short introduction will help you phrase your musical thoughts in terms the TI understands.

Basic Notation

In written music, different tones are represented by notes placed on a staff composed of five lines and four spaces. The two most common staves are the *treble* staff, used for middle to high tones, and the *bass* staff, used for middle to low tones.

Music is generally written on the *grand* staff, a combination of the bass and treble staves. The treble portion generally contains the melody, while the bass portion contains the harmony or accompaniment.

A note, by itself, simply indicates the duration of a tone. It is the position of that note on the staff that specifies a given tone, and each line or space is identified by one of the first seven letters of the alphabet. Figure 5-1 shows how the letter names and frequencies relate to the lines and spaces.

The interval from any note to the next one of the same letter name is called an *octave*. For example, the interval from the A with a frequency of 110 to the A with a frequency of 220 represents one octave. Each octave represents a doubling of frequency, and the following program demonstrates that relationship by playing each note and its octaves.

Program 5-2 will help you learn to identify notes by their letter names. It draws a grand staff on the screen and places labeled notes on the staff. As each note appears, CALL SOUND is used to generate the corresponding tone. After all the notes have been played, a short quiz reviews your note-naming skills.

Using symbols called *chromatic signs*, notes called *sharps* (#) or *flats* (b) can also be identified. A sharp will be slightly higher in frequency than the un-sharped note, while a flat will be slightly lower in frequency. The distance between two adjacent tones—for example, between C and C#—is called a *half step*; the distance between two tones with a single tone between them—C to D, for instance—is called a *whole step*. Chromatic signs which appear at the beginning of a piece are called the *key signature* and affect every note in the piece, while those appearing within the piece affect only a single measure.

The *natural* sign (n) may also be encountered. You can think of it as a “neutralizer” which does away with a sharp or flat. When placed in front of a note, it restores that note to its unmodified pitch. For example, even though the key signature might tell you to sharp every F, the natural sign can be used to selectively remove the sharp whenever desired.

Your TI can generate more than 44,000 tones, but only a few of them correspond to tones used in Western music. Table 5-2 lists those that do, along with the corresponding letter names used to represent them.

Figure 5-1. Notes on the Staff

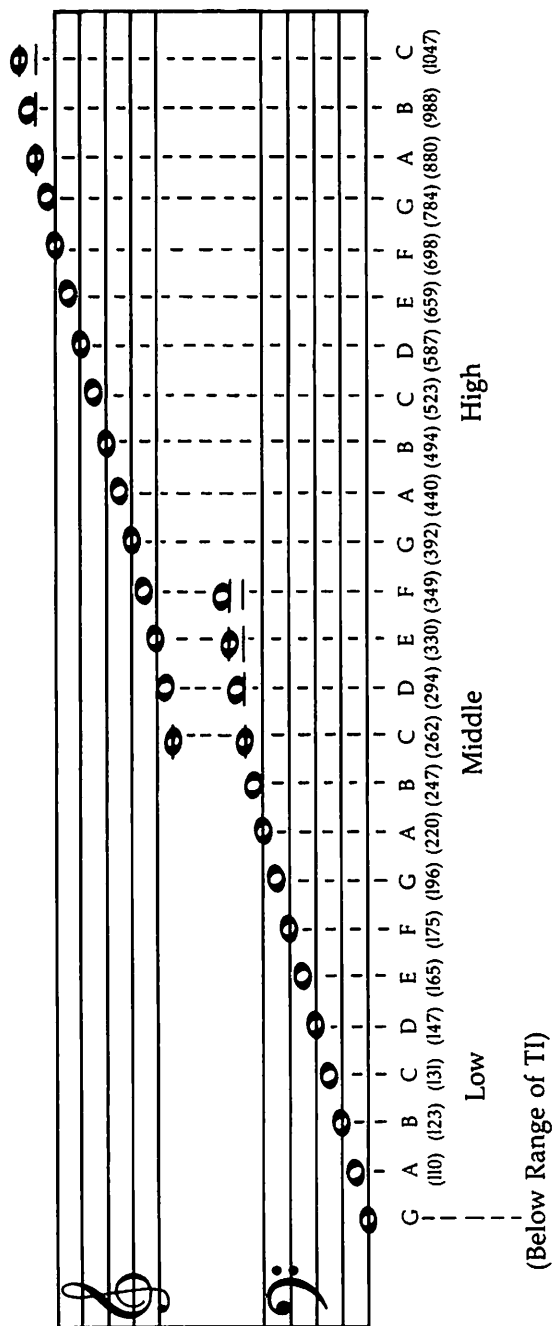














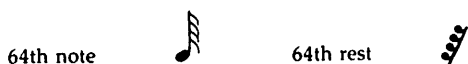
Table 5-2. Musical Tones Frequency

| Frequency | Note | Frequency | Note |
|-----------|--------------------|-----------|--------------------|
| 110 | A | 440 | A (above middle C) |
| 117 | A#, Bb | 466 | A#, Bb |
| 123 | B | 494 | B |
| 131 | C (low C) | 523 | C (high C) |
| 139 | C#, Db | 554 | C#, Db |
| 147 | D | 587 | D |
| 156 | D#, Eb | 622 | D#, Eb |
| 165 | E | 659 | E |
| 175 | F | 698 | F |
| 185 | F#, Gb | 740 | F#, Gb |
| 196 | G | 784 | G |
| 208 | G#, Ab | 831 | G#, Ab |
| 220 | A (below middle C) | 880 | A (above high C) |
| 233 | A#, Bb | 932 | A#, Bb |
| 247 | B | 988 | B |
| 262 | C (middle C) | 1047 | C |
| 277 | C#, Db | 1109 | C#, Db |
| 294 | D | 1175 | D |
| 311 | D#, Eb | 1245 | D#, Eb |
| 330 | E | 1319 | E |
| 349 | F | 1397 | F |
| 370 | F#, Gb | 1480 | F#, Gb |
| 392 | G | 1568 | G |
| 415 | G#, Ab | 1661 | G#, Ab |
| 440 | A (above middle C) | 1760 | A |

Besides indicating pitch, notes also indicate duration. Different notes represent different durations; other symbols, called *rests*, indicate specific pauses. Figure 5-2 shows the symbols used for various notes and rests, while Figure 5-3 shows how they relate to each other.

Figure 5-2. Notes and Rests

| | | | |
|--------------|---|--------------|---|
| Whole note |  | Whole rest |  |
| Half note |  | Half rest |  |
| Quarter note |  | Quarter rest |  |
| 8th note |  | 8th rest |  |
| 16th note |  | 16th rest |  |
| 32nd note |  | 32nd rest |  |

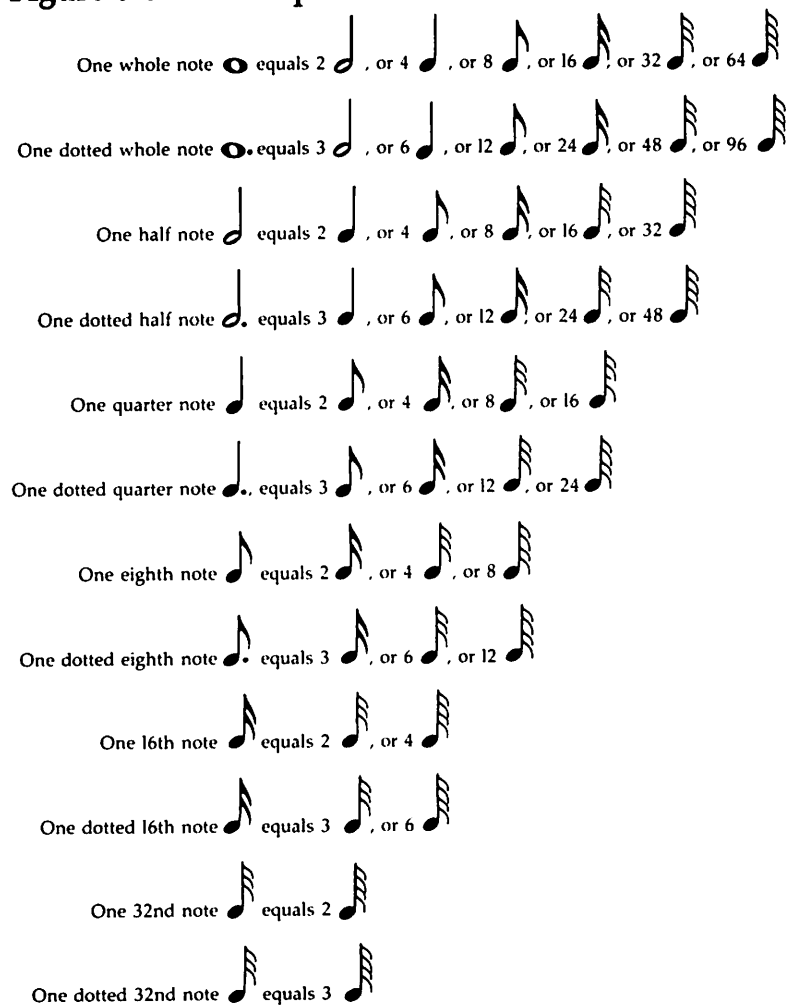


Note: Two or more 8th, 16th, 32nd, or 64th notes are usually connected by a beam. For example,



Note that the stems of notes may point down instead of up. The direction of the stem depends on the position of the note on the staff.

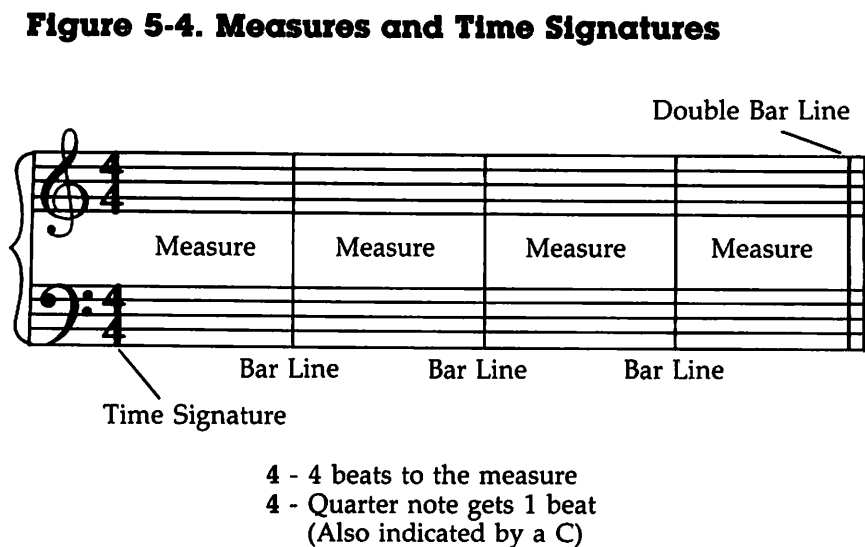
Figure 5-3. Note Equivalents



Note that different notes (and rests) represent relative rather than absolute durations. In other words, a whole note has no set length. But regardless of how long that whole note actually lasts, a half note will last half as long, a quarter note will last one-quarter as long, and so on.

A note can also be followed by a *dot*. The dot tells you to prolong that note by half its time value. For example, if a half note was followed by a dot, it would last three-fourths as long as a whole note ($1/2 + 1/4 = 3/4$).

Regardless of duration, notes are arranged in groups called measures. Measures divide music into segments with an equal number of beats. The lines used to delineate measures are called *bar lines*, and a double bar line is used at the end of a piece. Figure 5-4 shows a grand staff divided into measures.



Other Time Signatures

3 - 3 beats to the measure
 4 - Quarter note gets 1 beat

6 - 6 beats to the measure
 8 - Eighth note gets 1 beat

2 - 2 beats to the measure
 2 - Half note gets 1 beat

2 - 2 beats to the measure
 4 - Quarter note gets 1 beat

Information about the rhythm of a piece is given by the *time signature*, a numerical fraction found at the beginning of the grand staff. The top number tells you how many beats are in each measure, while the bottom number indicates the type of note that gets one beat. The time signature shown in Figure 5-4, for example, tells you that each measure contains four beats and that the quarter note should be counted as one beat. A half note would get two beats, and a whole note would get four beats. Similarly, an eighth note would get half a beat, a dotted half note would get three beats, and so on.

Program 5-1. Octaves

```

100 CALL CLEAR :: CALL SCREEN(2)
110 CALL MAGNIFY(2)
120 FOR L=1 TO 7
130 READ N(L)
140 NEXT L
150 FOR L=1 TO 7
160 F=N(L)
170 CALL SPRITE(#1,64+L,14,90,125)
180 FOR X=1 TO 5
190 CALL SOUND(1500,F,0)
200 F=F*2
210 NEXT X
220 FOR T=1 TO 1500
230 NEXT T
240 NEXT L
250 DATA 110,123,131,147,165,175,196

```

Program 5-2. Note Tutor

```

100 CALL CLEAR
110 RANDOMIZE
120 DIM NOTE$(22),ROW(22),FREQ(22),SW(22)
130 CALL CHAR(91,"000000FF00000000")
140 CALL CHAR(92,"01010101010101")
150 CALL CHAR(93,"8080808080808080")
160 CALL CHAR(96,"1866818181661800")
170 CALL CHAR(100,"010101FF01010101010101FF0305091
9313161FF47C9C9CDC5C7C3FF41616131")
180 CALL CHAR(104,"190D07FF01010101")
190 CALL CHAR(105,"C0A090FF0404040890A0C0FF8080808
0808080FFF0B88C86828181FF8383868E")
200 CALL CHAR(109,"98B0E0FF80808080")
210 CALL CHAR(110,"000000FF0F183060407038FF0000000
0000000FF00000000000000FF00000700")

```

Sound

```
220 CALL CHAR(114,"000000FF00000000")
230 CALL CHAR(115,"000000FFF01008080C0D0CFF0C0D0C0
    C0C0C0CFF0C0C0C08181030FF70E00
    000")
240 CALL CHAR(119,"000000FF00000000")
250 FOR L=1 TO 22
260 READ NOTE$(L),ROW(L),FREQ(L)
270 NEXT L
280 CALL CLEAR
290 DISPLAY AT(4,1):"1 - NOTE DRILL" :: DISPLAY AT
    (6,1):"2 - NOTE TEST"
300 DISPLAY AT(10,1):"SELECT ONE ---> " :: ACCEPT
    AT(10,17)VALIDATE("12")SIZE(-1)BEEP:OPT
310 CALL CLEAR
320 IF OPT=1 THEN DISPLAY AT(1,10):"D R I L L" ELS
    E DISPLAY AT(1,10):"T E S T"
330 FOR L=5 TO 9
340 CALL HCHAR(L,2,91,30)
350 NEXT L
360 FOR L=14 TO 18
370 CALL HCHAR(L,2,91,30)
380 NEXT L
390 FOR L=5 TO 9
400 CALL HCHAR(L,2,95+L,1)
410 CALL HCHAR(L,3,100+L,1)
420 NEXT L
430 FOR L=14 TO 18
440 CALL HCHAR(L,2,96+L,1)
450 CALL HCHAR(L,3,101+L,1)
460 NEXT L
470 CALL VCHAR(5,1,92,14):: CALL VCHAR(5,32,93,14)
480 DISPLAY AT(21,4):"LOW{3 SPACES}MIDDLE HIGH"
490 IF OPT=2 THEN 620
500 C=33 :: FOR L=1 TO 22
510 IF L=10 OR L=22 THEN CALL HCHAR(INT(ROW(L)/8)+
    1,3+L,91,3)
520 CALL SPRITE(#L,96,2,ROW(L),C)
530 C=C+8 :: DISPLAY AT(19,2+L):NOTE$(L)
540 CALL SOUND(800,FREQ(L),0)
550 FOR D=1 TO 1200 :: NEXT D
560 NEXT L
570 DISPLAY AT(23,5):"** DRILL COMPLETE **"
580 FOR D=1 TO 2500 :: NEXT D
590 CALL DELSPRITE(ALL)
600 GOTO 280
610 GOTO 610
620 FOR L=1 TO 22 :: SW(L)=0 :: NEXT L
630 CC=33 :: RIGHT=0 :: WRONG=0
```

```
640 DISPLAY AT(21,1):" "
650 FOR L=1 TO 10
660 N=1+INT(RND*22):: IF SW(N)=1 THEN 660
670 SW(N)=1
680 CALL SPRITE(#L,96,2,ROW(N),CC)
690 IF N=10 OR N=22 THEN CALL HCHAR(INT(ROW(N)/8)+
    1,3+L,91,3)
700 CALL SOUND(800,FREQ(N),0)
710 CC=CC+8
720 DISPLAY AT(23,1):"WHAT IS THIS NOTE? " :: ACC
    EPT AT(23,20)VALIDATE("ABCDEFGG")SIZE(-1):ANS$
730 CALL HCHAR(23,1,32,32)
740 IF ANS$=NOTE$(N)THEN DISPLAY AT(23,10):"R I G
    H T !" ELSE DISPLAY AT(23,10):"W R O N G !"
750 IF ANS$<>NOTE$(N)THEN DISPLAY AT(24,1):"THE CO
    RRECT ANSWER IS ";NOTE$(N):: WRONG=WRONG+1 ELS
    E RIGHT=RIGHT+1
760 DISPLAY AT(19,2+L):NOTE$(N)
770 FOR D=1 TO 1500 :: NEXT D
780 CALL HCHAR(23,1,32,32):: CALL HCHAR(24,1,32,32
    )
790 NEXT L
800 DISPLAY AT(23,1):"RIGHT: ";RIGHT;"{3 SPACES}WR
    ONG: ";WRONG
810 FOR D=1 TO 2500 :: NEXT D
820 CALL DELSPRITE(ALL)
830 GOTO 280
840 DATA "A",133,110
850 DATA "B",129,123
860 DATA "C",125,131
870 DATA "D",121,147
880 DATA "E",117,165
890 DATA "F",113,175
900 DATA "G",109,196
910 DATA "A",105,220
920 DATA "B",101,247
930 DATA "C",73,262
940 DATA "D",69,294
950 DATA "E",65,330
960 DATA "F",61,349
970 DATA "G",57,392
980 DATA "A",53,440
990 DATA "B",49,494
1000 DATA "C",45,523
1010 DATA "D",41,587
1020 DATA "E",37,659
1030 DATA "F",33,698
1040 DATA "G",29,784
1050 DATA "A",25,880
```

Making Music on the TI

Now that you've completed the crash course in basic music theory, you're ready to translate written music into the language of the TI. The simple piece shown in Figure 5-5, consisting only of a melody line, will be used as the first example.

Figure 5-5. Example Piece



The first thing to do is to determine the tempo of the piece. A tempo mark (generally an Italian word or phrase) usually appears at the top left corner of a musical piece. In this example, the tempo is marked *Moderato*, indicating that the music should be played at a moderate rate of speed. Some other commonly used tempo markings and their meanings are listed in Table 5-3.

Table 5-3. Tempo Marks In order of increasing speed

| | |
|----------------|-------------------------------------|
| Largo | broadly, very slowly |
| Lento | slowly |
| Adagio | slowly, leisurely |
| Andante | moderately slow, flowing |
| Andantino | slightly faster than andante |
| Moderato | moderately |
| Allegretto | quickly, but not as fast as allegro |
| Allegro | at a quick pace, lively |
| Vivace or Vivo | lively |
| Presto | very fast |
| Prestissimo | faster than presto |

The tempo lets you decide on the relative durations of individual notes. If the tempo indicated is fast, the notes would be shorter than if the tempo were slow. For example, in a piece marked *presto* a quarter note might last 250 milliseconds, while in a piece marked *lento* it would last 1000 milliseconds.

Next, one type of note must be chosen as the basis for determining the relative duration of the other notes in the piece. A good choice would be the note indicated in the time signature. This piece is written in 4/4 time; therefore, the quarter note is a logical choice.

Since the piece has a moderate tempo, the quarter note will be assigned a duration of 500 milliseconds. The durations of other types of notes could be computed separately. But it is simpler to assign the quarter note's duration to a variable and then express the durations of other notes in terms of that variable. For example, if T represents the duration of a quarter note, then a half note would be represented by $2 * T$, a whole note by $4 * T$, an eighth note by $T / 2$, and so on. This method also lets you change the tempo of the entire piece simply by changing the value of T .

Now you can determine the actual musical tones represented by the notes. The frequencies of those tones could be looked up note by note as the piece is transcribed; however, going back and forth between the music, the frequency table, and the keyboard can get rather tedious. The process is made easier by first determining the range of specific frequencies used in the piece and then assigning those frequencies to variables. The variable names should correspond to the letter names of the tones (that is, A, B, C, D, E, F, and G), and there must be some way of distinguishing tones with the same letter name. One variable-naming scheme for musical tones is shown in Table 5-4.

In this piece, the tones range from middle C to high C. However, before assigning frequencies to variable names, the key signature must be checked to see if any notes are sharped or flatted. This particular key signature indicates no sharps or flats, so no variables representing sharp or flat tones will be needed unless a sharp or flat appears in the body of the music.

Once the duration parameter has been established and the frequency variables have been defined, each note piece can be

translated into a CALL SOUND statement. "Music Demo 1" shows how this is done. For simplicity's sake, a single volume is used throughout the piece.

Table 5-4. Variable Names for Musical Tones

| Musical Tone | Variable Name | Musical Tone | Variable Name |
|----------------|---------------|---------------|---------------|
| A (110) | LLA | A (440) | A |
| A#, Bb (117) | LLAS or LLBF | A#, Bb (466) | AS or BF |
| B (123) | LLB | B (494) | B |
| Low C (131) | LC | High C (523) | HC |
| C#, Db (139) | LCS or LDF | C#, Db (554) | HCS or HDF |
| D (147) | LD | D (587) | HD |
| D#, Eb (156) | LDS or LEF | D#, Eb (622) | HDS or HEF |
| E (165) | LE | E (659) | HE |
| F (175) | LF | F (698) | HF |
| F#, Gb (185) | LFS or LGF | F#, Gb (740) | HFS or HGF |
| G (196) | LG | G (784) | HG |
| G#, Ab (208) | LGS or LAF | G#, Ab (831) | HGS or HAF |
| A (220) | LA | A (880) | HA |
| A#, Bb (233) | LAS or LBF | A#, Bb (932) | HAS or HBF |
| B (247) | LB | B (988) | HB |
| Middle C (262) | C | C (1047) | HHC |
| C#, Db (277) | CS or DF | C#, Db (1109) | HHCS or HHDF |
| D (294) | D | D (1175) | HHD |
| D#, Eb (311) | DS or EF | D#, Eb (1245) | HHDS or HHEF |
| E (330) | E | E (1319) | HHE |
| F (349) | F | F (1397) | HHF |
| F#, Gb (370) | FS or GF | F#, Gb (1480) | HHFS or HHGF |
| G (392) | G | G (1568) | HHG |
| G#, Ab (415) | GS or AF | G#, Ab (1661) | HHGS or HHAF |
| A (440) | A | A (1760) | HHA |

Program 5-3. Music Demo 1

```

100 T=500
110 C=262 :: D=294 :: E=330 :: F=349 :: G=392 :: A
    =440 :: B=494 :: HC=523
120 CALL SOUND(T,HC,0)
130 CALL SOUND(T/2,HC,0)
140 CALL SOUND(T/2,B,0)
150 CALL SOUND(T,A,0)
160 CALL SOUND(T,A,0)
170 CALL SOUND(T,G,0)
180 CALL SOUND(T/2,G,0)
190 CALL SOUND(T/2,F,0)

```

```

200 CALL SOUND(T,E,0)
210 CALL SOUND(T/2,E,0)
220 CALL SOUND(T/2,F,0)
230 CALL SOUND(T,G,0)
240 CALL SOUND(T,C,0)
250 CALL SOUND(T,D,0)
260 CALL SOUND(T,F,0)
270 CALL SOUND(T+T/2,E,0)
280 CALL SOUND(T/2,D,0)
290 CALL SOUND(T,C,0)

```

How Music Demo 1 Works

Line

- 100 Set the variable T to 500 (milliseconds). The variable T represents the duration of a quarter note.
- 110 Define the variables representing the frequencies of the notes from middle C to high C.
- 120 Translate the first note of the first measure into a CALL SOUND statement. This note is a quarter note representing the musical tone high C; therefore, its duration is represented by T, and its frequency is represented by HC.
- 130 Translate the second note of the first measure. This note is an eighth note (T/2) representing the musical tone high C (HC).
- 140 Translate the third note of the first measure. This note is an eighth note (T/2) representing the musical tone B.
- 150 Translate a quarter note (T) representing the tone A.
- 160 Translate a quarter note (T) representing the tone A.
- 170 Translate a quarter note (T) representing the tone G.
- 180 Translate an eighth note (T/2) representing the tone G.
- 190 Translate an eighth note (T/2) representing the tone F.
- 200 Translate a quarter note (T) representing the tone E.
- 210 Translate an eighth note (T/2) representing the tone E.
- 220 Translate an eighth note (T/2) representing the tone F.
- 230 Translate a quarter note (T) representing the tone G.
- 240 Translate a quarter note (T) representing the tone C.
- 250 Translate a quarter note (T) representing the tone D.
- 260 Translate a quarter note (T) representing the tone F.

- 270 Translate a dotted quarter note ($T + T/2$) representing the tone E.
- 280 Translate an eighth note ($T/2$) representing the tone D.
- 290 Translate a quarter note (T) representing the tone middle C.

Note that a quarter rest constitutes the last beat of the last measure. Had the rest appeared within the music, you would have needed a pause with the same duration as a quarter note (500 milliseconds). To get it, you could use a CALL SOUND statement specifying any frequency but the softest volume (30). For example, the following CALL SOUND statement would produce the pause called for by a quarter rest:

CALL SOUND (T,110,30)

Figure 5-6 expands the example piece to include bass accompaniment. Only minor program alterations are required. The relative note durations remain the same. However, the range of musical tones must be expanded since the bass notes range from low C to middle C and include two flats in the third measure. Once the variables for these additional tones have been defined, the CALL SOUND statements can be altered to include the bass notes.

Figure 5-6. Example Piece with Bass Accompaniment



When corresponding treble and bass notes have the same duration, as do the first treble and bass notes of the first measure, adding the bass note is simply a matter of adding another set of frequency and volume parameters. But if the bass and treble notes do not have the same duration, the duration parameter must be changed to correspond with the note

that is shorter. The longer note must be distributed over two (or more) CALL SOUND statements.

For example, the second treble note in the first measure is an eighth note, while its accompanying bass note is a quarter note. The CALL SOUND statement initiating both notes would have to have a duration parameter corresponding to an eighth note ($T/2$), which would complete the eighth note and the first half of the quarter note. The remainder of the quarter note would have to be completed in the next CALL SOUND statement—but it would still sound as a single, uninterrupted quarter note, even though it was distributed over two separate CALL SOUND statements.

The following program demonstrates how the bass accompaniment could be added to the example piece.

Program 5-4. Music Demo 2

```

100 T=500
110 LC=131 :: LD=147 :: LE=165 :: LF=175 :: LG=196
    :: LAF=208 :: LA=220 :: LBF=233 :: LB=247
120 C=262 :: D=294 :: E=330 :: F=349 :: G=392 :: A
    =440 :: B=494 :: HC=523
130 CALL SOUND(T,HC,0,LC,0)
140 CALL SOUND(T/2,HC,0,LE,0)
150 CALL SOUND(T/2,B,0,LE,0)
160 CALL SOUND(T,A,0,LF,0)
170 CALL SOUND(T,A,0,LF,0)
180 CALL SOUND(T,G,0,LG,0)
190 CALL SOUND(T/2,G,0,LB,0)
200 CALL SOUND(T/2,F,0,LB,0)
210 CALL SOUND(T,E,0,C,0)
220 CALL SOUND(T/2,E,0,C,0)
230 CALL SOUND(T/2,F,0,C,0)
240 CALL SOUND(T,G,0)
250 CALL SOUND(T,C,0,LBF,0)
260 CALL SOUND(T,D,0,LA,0)
270 CALL SOUND(T,F,0,LAF,0)
280 CALL SOUND(T,E,0,LG,0)
290 CALL SOUND(T/2,E,0,LF,0)
300 CALL SOUND(T/2,D,0,LF,0)
310 CALL SOUND(T,C,0,LE,0)

```

How Music Demo 2 Works

Line(s)



- 100 Set the variable T (duration of a quarter note) to 500.
- 110–120 Define the variables representing the frequencies of the notes.
- 130 The bass note, low C, is added. Since both notes are quarter notes, the duration remains the same.
- 140 Low E, a quarter note, is added. It must be included in the next CALL SOUND since its duration is longer than that of the eighth note it accompanies.
- 150 Low E is added again, and thus given its full duration.
- 160 Low F, a half note, is added. It must be included in the next CALL SOUND since its duration is longer than that of the quarter note it accompanies.
- 170 Low F is added again, and thus given its full duration.
- 180 Low G is added. Both notes are quarter notes.
- 190 Low B, a quarter note, is added. It must be included in the next CALL SOUND since its duration is longer than that of the eighth note it accompanies.
- 200 Low B is added again, and thus given its full duration.
- 210 Middle C, a half note, is added. It must be included in the next CALL SOUND since its duration is longer than that of the quarter note it accompanies.
- 220 Middle C is added again, but must also be included in the next CALL SOUND in order to be given its full duration.
- 230 Middle C is added again, and thus given its full duration.
- 240 A quarter rest accompanies a quarter note; therefore, no note is added.
- 250 Low B \flat is added. Both notes are quarter notes.
- 260 Low A is added. Both notes are quarter notes.
- 270 Low A \flat is added. Both notes are quarter notes.
- 280 Low G, a quarter note, is added. In this case, this bass note has a shorter duration than the treble note it accompanies; thus, the duration must be changed (T) and the treble note must be included in the next CALL SOUND statement.

- 290 The treble note, E, is continued for the remainder of its duration (T/2). Low F, a quarter note, is added. It must be included in the next CALL SOUND since its duration is longer than the remaining duration of the treble note it accompanies.
- 300 Low F is added again, and thus is given its full duration.
- 310 Low E is added. Both notes are quarter notes.

In the previous examples, both tempo and volume remained constant. However, by varying the appropriate parameters, the music can be given much more feeling. Most musical compositions contain expression marks indicating varying degrees of loudness as well as changes in tempo, and some commonly used expression marks are shown in Figure 5-7.

Figure 5-7. Expression Marks

Pertaining to Volume

| Sign | Italian Name | Meaning |
|---|------------------------------|-----------------------|
| pp | pianissimo | very soft |
| p | piano | soft |
| mp | mezzo piano | moderately soft |
| mf | mezzo forte | moderately loud |
| f | forte | loud |
| ff | fortissimo | very loud |
|  | crescendo (cresc.) | play gradually louder |
|  | decrescendo (decresc.) | play gradually softer |
| (No Sign) | diminuendo (dim., dimin.) | play gradually softer |

Pertaining to Tempo

| | | |
|----------------------|---|-----------------------|
| ritardando (rit.) | — | slow down |
| accelerando (accel.) | — | speed up |
| a tempo | — | resume original tempo |

The following program transcribes the musical piece shown in Figure 5-8. The volume and tempo are varied to give the piece expression.

Program 5-5. Music Demo 3

```

100 T=500
110 LD=147 :: LE=165 :: LFS=185 :: LG=196 :: LA=22
    Ø :: LB=247
120 C=262 :: D=294 :: E=330 :: FS=370 :: G=392 ::
    A=440 :: B=494
130 HC=523 :: HD=587 :: HE=659 :: HFS=740 :: HG=78
    4
140 CALL SOUND(T,HD,7,LB,7,LG,7)
150 CALL SOUND(T/2,G,7,LB,7,LG,7)
160 CALL SOUND(T/2,A,7,LB,7,LG,7)
170 CALL SOUND(T/2,B,7,LA,7)
180 CALL SOUND(T/2,HC,7,LA,7)
190 CALL SOUND(T,HD,7,LB,7)
200 CALL SOUND(T,G,7,LB,7)
210 CALL SOUND(T,G,7,LB,7)
220 CALL SOUND(T,HE,6,C,6)
230 CALL SOUND(T/2,HC,5,C,5)
240 CALL SOUND(T/2,HD,4,C,4)
250 CALL SOUND(T/2,HE,3,C,3)
260 CALL SOUND(T/2,HFS,2,C,2)
270 CALL SOUND(T,HG,Ø,LB,Ø)
280 CALL SOUND(T,G,Ø,LB,Ø)
290 CALL SOUND(T,G,Ø,LB,Ø)
300 CALL SOUND(T,HC,Ø,LA,Ø)
310 CALL SOUND(T/2,HD,Ø,LA,Ø)
320 CALL SOUND(T/2,HC,Ø,LA,Ø)
330 CALL SOUND(T/2,B,Ø,LA,Ø)
340 CALL SOUND(T/2,A,Ø,LA,Ø)
350 CALL SOUND(T,B,Ø,LG,Ø)
360 CALL SOUND(T/2,HC,Ø,LG,Ø)
370 CALL SOUND(T/2,B,Ø,LG,Ø)
380 CALL SOUND(T/2,A,Ø,LG,Ø)
390 CALL SOUND(T/2,G,Ø,LG,Ø)
400 CALL SOUND(T,A,2,C,2)
410 T=700
420 CALL SOUND(T/2,B,3,D,3)
430 CALL SOUND(T/2,A,4,D,4)
440 CALL SOUND(T/2,G,5,LD,5)
450 CALL SOUND(T/2,FS,6,LD,6)
460 CALL SOUND(3*T,G,7,LG,7)

```

Figure 5-8. Music for Music Demo 3

The image displays two systems of musical notation for a piano and bass. The first system is marked "Moderato" and "mp" (mezzo-piano). The piano staff (top) begins with a series of eighth notes, followed by a phrase of eighth notes, and then a series of quarter notes. The bass staff (bottom) features a series of quarter notes, followed by a series of eighth notes, and then a series of quarter notes. The second system continues the piece, with the piano staff featuring a series of eighth notes, followed by a phrase of eighth notes, and then a series of quarter notes. The bass staff features a series of quarter notes, followed by a series of eighth notes, and then a series of quarter notes. The notation includes various musical symbols such as clefs, key signatures, time signatures, and dynamic markings.

How Music Demo 3 Works

Line(s)

- 100 This piece has a moderate tempo, and its time signature indicates that the quarter note would be a good basis for the duration parameter. Therefore, the variable T represents the duration of a quarter note and is set to 500 milliseconds.
- 110–130 Define the variables representing the frequencies of the notes. Notice that the key signature indicates that all F's are sharped throughout the piece.
- 140 High D (quarter note), low B (half note), and low G (half note) are transcribed. The shorter duration of the quarter note (T) is specified. The volume of the notes is set to 7 since the expression mark (mp) indicates that they are to be played moderately softly.
- 150 G (eighth note) is transcribed. Low B and low G are continued. The shorter duration of G is continued. The shorter duration of G (T/2) is used.
- 160 A (eighth note) is transcribed. Low B and low G are continued for the remainder of their duration (T/2), which is the same as the duration of A.
- 170 B (eighth note) and low A (quarter note) are transcribed. The shorter duration of B (T/2) is used.
- 180 High C (eighth note) is transcribed. Low A is continued for the remainder of its duration (T/2), which is the same as the duration of high C.
- 190 High D (quarter note) and low B (dotted half note) are transcribed. The shorter duration of high D (T) is used.
- 200 G (quarter note) is transcribed. Low B is continued. The shorter duration of G (T) is used.
- 210 G (quarter note) is transcribed. Low B is continued for the remainder of its duration (T), which is the same as the duration of G.
- 220 High E (quarter note) and middle C (dotted half note) are transcribed. The shorter duration of high E (T) is used. The expression mark indicates that the notes are to be played gradually louder. Therefore, the volume specified is gradually increased.
- 230 High C (eighth note) is transcribed. Middle C is continued. The shorter duration of high C (T/2) is used.

- 240 High D (eighth note) is transcribed. Middle C is continued. The shorter duration of high D ($T/2$) is used.
- 250 High E (eighth note) is transcribed. Middle C is continued. The shorter duration of high E is used.
- 260 High F sharp (eighth note) is transcribed. Middle C is continued for the remainder of its duration ($T/2$), which is the same as the duration of high F sharp.
- 270 High G (quarter note) and low B (dotted half note) are transcribed. The shorter duration of high G (T) is used.
- 280 G (quarter note) is transcribed. Low B is continued. The shorter duration of G (T) is used.
- 290 G (quarter note) is transcribed. Low B is continued for the remainder of its duration (T), which is the same as the duration of G.
- 300 High C (quarter note) and low A (dotted half note) are transcribed. The shorter duration of high C (T) is used.
- 310 High D (eighth note) is transcribed. Low A is continued. The shorter duration of high D ($T/2$) is used.
- 320 High C (eighth note) is transcribed. Low A is continued. The shorter duration of high C is used.
- 330 B (eighth note) is transcribed. Low A is continued. The shorter duration of B is used.
- 340 A (eighth note) is transcribed. Low A is continued for the remainder of its duration ($T/2$), which is the same as the duration of A.
- 350 B (quarter note) and low G (dotted half note) are transcribed. The shorter duration of B (T) is used.
- 360 High C (eighth note) is transcribed. Low G is continued. The shorter duration of high C ($T/2$) is used.
- 370 B (eighth note) is transcribed. Low G is continued. The shorter duration of B ($T/2$) is used.
- 380 A (eighth note) is transcribed. Low G is continued. The shorter duration of A is used.
- 390 G (eighth note) is transcribed. Low G is continued for the remainder of its duration ($T/2$), which is the same as the duration of G.
- 400 A and middle C are transcribed. Both notes are quarter notes (T). The expression mark indicates that the notes should be played gradually softer; therefore, the volume specified is gradually decreased.

Sound

- 410 The expression mark (rit.) indicates that the notes should be played slower; therefore, the basis for the duration parameter is lengthened.
- 420 B (eighth note) and D (quarter note) are transcribed. The shorter duration of B ($T/2$) is used.
- 430 A (eighth note) is transcribed. D is continued for the remainder of its duration ($T/2$), which is the same as the duration of A.
- 440 G (eighth note) and low D (quarter note) are transcribed. The shorter duration of G ($T/2$) is used.
- 450 F-sharp (eighth note) is transcribed. Low D is continued for the remainder of its duration ($T/2$), which is the same as the duration of F-sharp.
- 460 G and low G are transcribed. Both notes are dotted half notes ($3 \times T$).

The final program in this section is the transcription of a fairly complex Bach prelude. One problem which may arise when working with such a piece is the need to play more than three notes at one time. In such a case, one of the notes will have to be excluded. Which to exclude is best decided by trying the possible combinations and letting your ear be the judge. Another problem may be caused by notes below the range of the TI. This can sometimes be handled by replacing the note with the same note an octave higher; if that doesn't sound right, the note will have to be excluded.

Program 5-6. Bach Prelude

```
100 T=500
110 LLA=110 :: LLBF=117 :: LC=131 :: LD=147 :: LE=
    165 :: LF=175 :: LG=196 :: LA=
220 :: LBF=233
120 C=262 :: D=294 :: E=330 :: F=349 :: G=392 :: A
    =440 :: BF=466
130 HC=523 :: HD=587 :: HE=659 :: HF=698 :: HG=784
    :: HA=880 :: HBF=932
140 CALL CLEAR
150 CALL SCREEN(14)
160 DISPLAY AT(5,9):"P R E L U D E"
170 DISPLAY AT(9,9):"J. S. B A C H"
180 FOR DEL=1 TO 1000 :: NEXT DEL
190 CALL SOUND(T/4,LA,10,LF,10)
200 CALL SOUND(T/4,HC,10)
```

210 CALL SOUND(T/4,A,10,C,10,LF,10)
220 CALL SOUND(T/4,HC,10)
230 CALL SOUND(T/4,F,9,LA,9,LF,9)
240 CALL SOUND(T/4,HC,9)
250 CALL SOUND(T/4,A,9,C,9,LF,9)
260 CALL SOUND(T/4,HC,9)
270 CALL SOUND(T/4,F,8)
280 CALL SOUND(T/4,HD,8)
290 CALL SOUND(T/4,BF,8,D,8,LF,8)
300 CALL SOUND(T/4,HD,8)
310 CALL SOUND(T/4,F,7,LBF,7,LF,7)
320 CALL SOUND(T/4,HD,7)
330 CALL SOUND(T/4,BF,7,D,7,LF,7)
340 CALL SOUND(T/4,HD,7)
350 CALL SOUND(T/4,G,6)
360 CALL SOUND(T/4,HE,6)
370 CALL SOUND(T/4,BF,6,LG,6,LF,6)
380 CALL SOUND(T/4,HE,6)
390 CALL SOUND(T/4,G,6,LBF,6,LF,6)
400 CALL SOUND(T/4,HE,6)
410 CALL SOUND(T/4,BF,6,LG,6,LF,6)
420 CALL SOUND(T/4,HE,6)
430 CALL SOUND(T/4,HF,7,LA,7,LF,7)
440 CALL SOUND(T/4,HC,8)
450 CALL SOUND(T/4,A,9)
460 CALL SOUND(T/4,HC,10)
470 CALL SOUND(T/4,F,10)
480 CALL SOUND(T/4,C,10)
490 CALL SOUND(T/4,LA,10)
500 CALL SOUND(T/4,C,10)
510 CALL SOUND(T/4,LF,10)
520 CALL SOUND(T/4,C,10)
530 CALL SOUND(T/4,HF,9,HC,9,LA,9)
540 CALL SOUND(T/4,C,9)
550 CALL SOUND(T/4,HF,9,A,9,LF,9)
560 CALL SOUND(T/4,C,9)
570 CALL SOUND(T/4,HF,8,HC,8,LA,8)
580 CALL SOUND(T/4,C,8)
590 CALL SOUND(T/4,LF,8)
600 CALL SOUND(T/4,D,8)
610 CALL SOUND(T/4,HF,7,HD,7,LBF,7)
620 CALL SOUND(T/4,D,7)
630 CALL SOUND(T/4,HF,7,BF,7,LF,7)
640 CALL SOUND(T/4,D,7)
650 CALL SOUND(T/4,HF,6,HD,6,LBF,6)
660 CALL SOUND(T/4,D,6)
670 CALL SOUND(T/4,LG,6)
680 CALL SOUND(T/4,E,6)
690 CALL SOUND(T/4,HG,6,HE,6,LBF,6)

Sound

700 CALL SOUND(T/4,E,6)
710 CALL SOUND(T/4,HBF,6,HE,6,LG,6)
720 CALL SOUND(T/4,E,6)
730 CALL SOUND(T/4,HG,6,HE,6,LBF,6)
740 CALL SOUND(T/4,E,6)
750 CALL SOUND(T/4,HA,7,HF,7,F,7)
760 CALL SOUND(T/4,C,7)
770 CALL SOUND(T/4,LA,8)
780 CALL SOUND(T/4,C,8)
790 CALL SOUND(T/4,LF,9)
800 CALL SOUND(T/4,LC,9)
810 CALL SOUND(T/4,LLA,10)
820 CALL SOUND(T/4,LC,10)
830 CALL SOUND(T/4,LF,10)
840 CALL SOUND(T/4,HA,10)
850 CALL SOUND(T/4,HF,10,LG,10)
860 CALL SOUND(T/4,HA,10)
870 CALL SOUND(T/4,HC,9,LLA,9)
880 CALL SOUND(T/4,HA,9)
890 CALL SOUND(T/4,HF,9,LBF,9)
900 CALL SOUND(T/4,HA,9)
910 CALL SOUND(T/4,HC,8,LC,8)
920 CALL SOUND(T/4,HG,8)
930 CALL SOUND(T/4,HF,8,LD,8)
940 CALL SOUND(T/4,HC,8)
950 CALL SOUND(T/4,HC,7,LE,7)
960 CALL SOUND(T/4,HG,7)
970 CALL SOUND(T/4,HE,7,LC,7)
980 CALL SOUND(T/4,HG,7)
990 CALL SOUND(T/4,A,6,LD,6)
1000 CALL SOUND(T/4,HF,6)
1010 CALL SOUND(T/4,HE,6,LE,6)
1020 CALL SOUND(T/4,HF,6)
1030 CALL SOUND(T/4,A,6,LF,6)
1040 CALL SOUND(T/4,HF,6)
1050 CALL SOUND(T/4,HD,6,LG,6)
1060 CALL SOUND(T/4,HF,6)
1070 CALL SOUND(T/4,A,6,LA,6)
1080 CALL SOUND(T/4,HE,6)
1090 CALL SOUND(T/4,HD,6,LBF,6)
1100 CALL SOUND(T/4,HE,6)
1110 CALL SOUND(T/4,A,6,C,6)
1120 CALL SOUND(T/4,HE,6)
1130 CALL SOUND(T/4,HC,6,LA,6)
1140 CALL SOUND(T/4,HE,6)
1150 CALL SOUND(T/4,F,6,LBF,6)
1160 CALL SOUND(T/4,HD,6)
1170 CALL SOUND(T/4,HC,6,LA,6)
1180 CALL SOUND(T/4,HD,6)

1190 CALL SOUND(T/4,G,6,LG,6)
1200 CALL SOUND(T/4,HD,6)
1210 CALL SOUND(T/4,BF,6,LF,6)
1220 CALL SOUND(T/4,HD,6)
1230 CALL SOUND(T/4,G,6,LE,6)
1240 CALL SOUND(T/4,HC,6)
1250 CALL SOUND(T/4,BF,6,LC,6)
1260 CALL SOUND(T/4,HC,6)
1270 CALL SOUND(T/4,F,6,LF,6)
1280 CALL SOUND(T/4,HC,6)
1290 CALL SOUND(T/4,A,6,LE,6)
1300 CALL SOUND(T/4,HC,6)
1310 CALL SOUND(T/4,F,6,LD,6)
1320 CALL SOUND(T/4,BF,6)
1330 CALL SOUND(T/4,A,6,LG,6)
1340 CALL SOUND(T/4,BF,6)
1350 CALL SOUND(T/4,C,6,LE,6)
1360 CALL SOUND(T/4,BF,6)
1370 CALL SOUND(T/4,G,6,LC,6)
1380 CALL SOUND(T/4,BF,6)
1390 CALL SOUND(T/4,C,6,LF,6)
1400 CALL SOUND(T/4,A,6,LC,6)
1410 CALL SOUND(T/4,F,6,LLA,6)
1420 CALL SOUND(T/4,A,6,LC,6)
1430 CALL SOUND(T/4,C,6,LF,6)
1440 CALL SOUND(T/4,A,6,LLA,6)
1450 CALL SOUND(T/4,F,6,LC,6)
1460 CALL SOUND(T/4,A,6,LF,6)
1470 CALL SOUND(T/4,D,6,LLBF,6)
1480 CALL SOUND(T/4,A,6)
1490 CALL SOUND(T/4,F,6,LBF,6)
1500 CALL SOUND(T/4,A,6)
1510 CALL SOUND(T/4,D,6)
1520 CALL SOUND(T/4,F,6)
1530 CALL SOUND(T/4,A,6,LA,6)
1540 CALL SOUND(T/4,HC,6)
1550 CALL SOUND(T/4,D,10,LG,10)
1560 CALL SOUND(T/4,BF,10,LD,10)
1570 CALL SOUND(T/4,G,9,LLBF,9)
1580 CALL SOUND(T/4,BF,9,LD,9)
1590 CALL SOUND(T/4,D,8,LG,8)
1600 CALL SOUND(T/4,BF,7,LLBF,7)
1610 CALL SOUND(T/4,G,6,LD,6)
1620 CALL SOUND(T/4,BF,6,LG,6)
1630 CALL SOUND(T/4,E,5,LC,5)
1640 CALL SOUND(T/4,BF,5)
1650 CALL SOUND(T/4,G,6,C,6)
1660 CALL SOUND(T/4,BF,6)
1670 CALL SOUND(T/4,E,7)

Sound

1680 CALL SOUND(T/4,G,7)
1690 CALL SOUND(T/4,BF,7,LBF,7)
1700 CALL SOUND(T/4,HD,7)
1710 CALL SOUND(T/4,F,7,LA,7)
1720 CALL SOUND(T/4,HC,7)
1730 CALL SOUND(T/4,A,7,LC,7)
1740 CALL SOUND(T/4,HC,7)
1750 CALL SOUND(T/4,HF,7,LA,7)
1760 CALL SOUND(T/4,HC,7)
1770 CALL SOUND(T/4,A,7,LC,7)
1780 CALL SOUND(T/4,F,7)
1790 CALL SOUND(T/4,LG,10)
1800 CALL SOUND(T/4,BF,10)
1810 CALL SOUND(T/4,G,10,LC,10)
1820 CALL SOUND(T/4,BF,10)
1830 CALL SOUND(T/4,HE,10,LG,10)
1840 CALL SOUND(T/4,BF,10)
1850 CALL SOUND(T/4,G,10,LC,10)
1860 CALL SOUND(T/4,E,10)
1870 CALL SOUND(T/4,LF,10)
1880 CALL SOUND(T/4,A,10)
1890 CALL SOUND(T/4,F,10,LC,10)
1900 CALL SOUND(T/4,A,10)
1910 CALL SOUND(T/4,HD,10,LF,10)
1920 CALL SOUND(T/4,A,10)
1930 CALL SOUND(T/4,F,10,LC,10)
1940 CALL SOUND(T/4,D,10)
1950 CALL SOUND(T/4,LE,10)
1960 CALL SOUND(T/4,G,10)
1970 CALL SOUND(T/4,E,10,LC,10)
1980 CALL SOUND(T/4,G,10)
1990 CALL SOUND(T/4,HC,10,LE,10)
2000 CALL SOUND(T/4,G,10)
2010 CALL SOUND(T/4,E,10,LC,10)
2020 CALL SOUND(T/4,C,10)
2030 CALL SOUND(T/4,LD,10)
2040 CALL SOUND(T/4,F,10)
2050 CALL SOUND(T/4,D,10,LC,10)
2060 CALL SOUND(T/4,F,10)
2070 CALL SOUND(T/4,G,10,LLBF,10)
2080 CALL SOUND(T/4,A,10)
2090 CALL SOUND(T/4,BF,10,LG,10)
2100 CALL SOUND(T/4,D,10)
2110 CALL SOUND(T/4,G,10,LE,10)
2120 CALL SOUND(T/4,A,10)
2130 CALL SOUND(T/4,BF,10,LC,10)
2140 CALL SOUND(T/4,E,10)
2150 CALL SOUND(T/4,F,10,LA,10)
2160 CALL SOUND(T/4,G,10)

```

2170 CALL SOUND(T/4,A,10,LF,10)
2180 CALL SOUND(T/4,C,10)
2190 CALL SOUND(T/4,D,9,LBF,9)
2200 CALL SOUND(T/4,F,9)
2210 CALL SOUND(T/4,A,8)
2220 CALL SOUND(T/4,HC,8)
2230 CALL SOUND(T/4,BF,7)
2240 CALL SOUND(T/4,A,7)
2250 CALL SOUND(T/4,G,6)
2260 CALL SOUND(T/4,BF,6)
2270 CALL SOUND(T/4,HD,5)
2280 CALL SOUND(T/4,HF,5)
2290 CALL SOUND(T/4,HE,4)
2300 CALL SOUND(T/4,HD,4)
2310 CALL SOUND(T/4,HC,3)
2320 CALL SOUND(T/4,HE,3)
2330 CALL SOUND(T/4,HG,2)
2340 CALL SOUND(T/4,HBF,2)
2350 CALL SOUND(T,HBF,0,HG,0,LE,0)
2360 CALL SOUND(T/2,HBF,30)
2370 CALL SOUND(T/2,HA,0,HF,0,LF,0)
2380 CALL SOUND(T/4,BF,0,LD,0)
2390 CALL SOUND(T/4,HG,0,HE,0)
2400 CALL SOUND(T/4,A,0,LLBF,0)
2410 CALL SOUND(T/4,HF,0,HD,0)
2420 CALL SOUND(T/4,G,0,LC,0)
2430 CALL SOUND(T/4,HF,0,HC,0)
2440 CALL SOUND(T/4,G,0,LC,0)
2450 CALL SOUND(T/4,HE,0,BF,0)
2460 CALL SOUND(T*2,HF,0,HC,0,LF,0)
2470 FOR X=1 TO 500
2480 NEXT X
2490 CALL CLEAR
2500 PRINT "PLAY IT AGAIN?"
2510 PRINT
2520 PRINT "ENTER Y OR N:"
2530 CALL KEY(3,REPLY,STATUS)
2540 IF STATUS=0 THEN 2530
2550 IF REPLY=89 THEN 140
2560 IF REPLY<>78 THEN 2500
2570 END

```

Sound Effects

As you'll recall, a sound's character is determined by its vibrational pattern, frequency, and amplitude. That is true whether the sound is music, footsteps, or the blast of a rocket engine.

Not surprisingly, a great many sounds other than musical

Sound

tones can be created by using the CALL SOUND subprogram. The examples that follow demonstrate some of these sounds. Keep in mind, however, that these examples represent only a small fraction of the sounds the TI is capable of producing. You will discover hundreds of others by experimenting with CALL SOUND.

Rocket in Motion

The sound of a rocket's engines can easily be produced by using noise frequency -7. For example, a continuous burn can be simulated by placing a CALL SOUND statement in a loop, as in the following example.

```
100 REM ROCKET IN MOTION
110 FOR L=1 to 10
120 CALL SOUND(1000,-7,3)
130 NEXT L
```

Morse Code

You can simulate a Morse code transmitter by using the RND function to alternate between two durations. In the following program, the 700 frequency is sounded for either 40 milliseconds (short) or 150 milliseconds (long). The loop in line 130 introduces a slight delay so that the tones have time to complete.

```
100 REM MORSE CODE
110 RANDOMIZE
120 IF RND>.5 THEN CALL SOUND(150,700,0) ELSE CALL
    SOUND(40,700,0)
130 FOR D=1 TO 20 :: NEXT D
140 GOTO 120
```

Computer

Movies about computers are generally accompanied by an impressionistic conception of what a computer might sound like. The following program example produces one such sound. It randomly picks a frequency between 2500 and 4750, in increments of 250, and plays it for 15 milliseconds.

```
100 REM COMPUTER
110 RANDOMIZE
120 FOR L=1 to 200
130 CALL SOUND(15,2500+(250*(RND)),0)
140 NEXT L
```

Sirens

Sirens used by American police are generally identified by a rising, and then falling, pitch. The first program uses two loops to produce such a sound. The first loop increases the frequency each time the CALL SOUND statement is encountered, while the second loop decreases the frequency for each CALL SOUND.

Notice that a negative duration is used. This causes the last CALL SOUND to end as soon as the next one is encountered. The result is a smoothly rising or falling tone. The choice of 60 for duration was somewhat arbitrary, since any negative duration immediately terminates the previous CALL SOUND. Generally, any number above 50 will work. A number below 50 produces a tone that will finish before BASIC can complete the loop. In this case, the result is a gap in the sound.

```
100 REM AMERICAN SIREN
110 FOR T=1 TO 10
120 FOR L=800 to 1200 STEP 8
130 CALL SOUND(-60,L,0)
140 NEXT L
150 FOR L=1200 to 800 STEP -8
160 CALL SOUND(-60,L,0)
170 NEXT L
180 NEXT T
```

To emulate a European-style siren, which alternates between two distinct frequencies, RUN the following program.

```
100 REM EUROPEAN SIREN
110 FOR L=1 to 10
120 CALL SOUND(400,500,0)
130 CALL SOUND(400,300,0)
140 NEXT L
```

Bomb and Explosion

You can simulate a falling bomb by combining decreasing pitch with decreasing volume. Then use a low frequency (110) and a -7 noise frequency, again with decreasing volume, to produce the explosion. Negative duration was used to produce a smooth decrease in pitch.

```
100 REM BOMB AND EXPLOSION
110 FOR L=1 to 90
```



```

120 CALL SOUND(-60,1400-(L*6),L/9)
130 NEXT L
140 FOR L=0 to 30
150 CALL SOUND(-60,110,L,-7,L)
160 NEXT L

```

Bells

When a bell is struck, several things occur. The bell vibrates at its primary frequency, it vibrates at its secondary frequencies (overtones), and its amplitude (volume) gradually decreases.

The next program imitates that sound, and the result is unmistakably that of a bell. By using the three voices available on the TI to produce the primary and secondary frequencies, and by using a loop to produce a decaying amplitude, the bell sound can easily be simulated with CALL SOUND.

```

100 REM BELL
110 FOR L=0 to 30 STEP 2
120 CALL SOUND(-50,700,L,2100,L,4200,L)
130 NEXT L

```

By lowering the frequencies and adding a second loop, a clock-type bell can be produced.

```

100 CLOCK BELL
110 FOR L=1 TO 6
120 FOR L2=0 TO 30 STEP 3
130 CALL SOUND(-50,400,L2,1200,L2,2400,L2)
140 NEXT L2
150 FOR D=1 TO 200 :: NEXT D
160 NEXT L

```

6

Speech Synthesis

Speech Synthesis

You've already seen that graphics, music, and sound effects can mean the difference between an average program and an outstanding one. But another impressive program addition is speech.

The possibilities for using computer-generated speech are virtually endless. It can be incorporated into a game program to verbally offer instructions, for instance, or to inject comments (good or bad) on the progress or skill of the players. In a tutorial program, speech can be used to literally *tell* a student if an answer is correct—and, if the answer was wrong, the speech synthesizer can give the correct answer. Speech also permits verbal explanations in educational programs for children who haven't yet learned to read. Besides, apart from being useful in programs, it's fun to have a computer that can talk.

In order to produce speech on your TI, you must have TI's PHP-1500 speech synthesizer peripheral. This device incorporates the Texas Instruments TMS5200 speech synthesis processor chip, which was first introduced in 1978 as part of TI's *Speak and Spell* product line. The TMS5200 uses linear predictive coding (LPC) techniques to generate a mathematical model of human speech from the incoming text string. The resulting electronic vocal tract is passed through a digital-to-analog converter, which yields an audio signal, and eventually on to the speaker.

In Extended BASIC, there are two ways to add speech to your programs. One is to use the speech synthesizer's resident vocabulary (Table 6-1) which contains 373 letters, numbers, words, and phrases. This vocabulary can be enlarged by manipulating the speech codes that make up the words.

Alternatively, you can use TI's text-to-speech diskette software, which is designed to work with TI's Extended BASIC command module. It gives you the capability to synthesize almost any word in the English language.

Table 6-1. Resident Vocabulary

| | | |
|--------------|-----------|----------|
| - (NEGATIVE) | BOTTOM | DOWN |
| + (POSITIVE) | BUT | DRAW |
| . (POINT) | BUY | DRAWING |
| 0 | BY | |
| 1 | BYE | E |
| 2 | | EACH |
| 3 | C | EIGHT |
| 4 | CAN | EIGHTY |
| 5 | CASSETTE | ELEVEN |
| 6 | CENTER | ELSE |
| 7 | CHECK | END |
| 8 | CHOICE | ENDS |
| 9 | CLEAR | ENTER |
| | COLOR | ERROR |
| A | COME | EXACTLY |
| A (a) | COMES | EYE |
| A1 (uh) | COMMA | |
| ABOUT | COMMAND | F |
| AFTER | COMPLETE | FIFTEEN |
| AGAIN | COMPLETED | FIFTY |
| ALL | COMPUTER | FIGURE |
| AM | CONNECTED | FIND |
| AN | CONSOLE | FINE |
| AND | CORRECT | FINISH |
| ANSWER | COURSE | FINISHED |
| ANY | CYAN | FIRST |
| ARE | | FIT |
| AS | D | FIVE |
| ASSUME | DATA | FOR |
| AT | DECIDE | FORTY |
| | DEVICE | FOUR |
| B | DID | FOURTEEN |
| BACK | DIFFERENT | FOURTH |
| BASE | DISKETTE | FROM |
| BE | DO | FRONT |
| BETWEEN | DOES | |
| BLACK | DOING | G |
| BLUE | DONE | GAMES |
| BOTH | DOUBLE | GET |

GETTING
GIVE
GIVES
GO
GOES
GOING
GOOD
GOOD WORK
GOODBYE
GOT
GRAY
GREEN
GUESS

H
HAD
HAND
HANDHELD UNIT
HAS
HAVE
HEAD
HEAR
HELLO
HELP
HERE
HIGHER
HIT
HOME
HOW
HUNDRED
HURRY

I
I WIN
IF
IN
INCH
INCHES
INSTRUCTION
INSTRUCTIONS
IS
IT

J
JOYSTICK
JUST

K
KEY
KEYBOARD
KNOW

L
LARGE
LARGER
LARGEST
LAST
LEARN
LEFT
LESS
LET
LIKE
LIKES
LINE
LOAD
LONG
LOOK
LOOKS
LOWER

M
MADE
MAGENTA
MAKE
ME
MEAN
MEMORY
MESSAGE
MESSAGES
MIDDLE
MIGHT
MODULE
MORE
MOST
MOVE

MUST

N
NAME
NEAR
NEED
NEGATIVE
NEXT
NICE TRY
NINE
NINETY
NO
NOT
NOW
NUMBER

O
OF
OFF
OH
ON
ONE
ONLY
OR
ORDER
OTHER
OUT
OVER

P
PART
PARTNER
PARTS
PERIOD
PLAY
PLAYS
PLEASE
POINT
POSITION
POSITIVE
PRESS
PRINT

PRINTER
PROBLEM
PROBLEMS
PROGRAM
PUT
PUTTING

Q

R
RANDOMLY
READ (read)
READ1 (red)
READY TO START
RECORDER
RED
REFER
REMEMBER
RETURN
REWIND
RIGHT
ROUND

S

SAID
SAVE
SAY
SAYS
SCREEN
SECOND
SEE
SEES
SET
SEVEN
SEVENTY
SHAPE
SHAPES
SHIFT
SHORT
SHORTER
SHOULD
SIDE

SIDES
SIX
SIXTY
SMALL
SMALLER
SMALLEST
SO

SOME
SORRY
SPACE
SPACES
SPELL
SQUARE
START
STEP
STOP
SUM
SUPPOSED
SUPPOSED TO
SURE

T

TAKE
TEEN
TELL
TEN
TEXAS INSTRUMENTS
THAN
THAT
THAT IS INCORRECT
THAT IS RIGHT
THE (the)
THE1 (thuh)
THEIR
THEN
THERE
THESE
THEY
THING
THINGS
THINK
THIRD

THIRTEEN
THIRTY
THIS
THREE
THREW
THROUGH
TIME
TO
TOGETHER
TONE
TOO
TOP
TRY
TRY AGAIN
TURN
TWELVE
TWENTY
TWO
TYPE

U

UHOH
UNDER
UNDERSTAND
UNTIL
UP
UPPER
USE

V

VARY
VERY

W

WAIT
WANT
WANTS
WAY
WE
WEIGH
WEIGHT
WELL

| | |
|---------------|---------|
| WERE | X |
| WHAT | |
| WHAT WAS THAT | Y |
| WHEN | YELLOW |
| WHERE | YES |
| WHICH | YET |
| WHITE | YOU |
| WHO | YOU WIN |
| WHY | YOUR |
| WILL | |
| WITH | Z |
| WON | ZERO |
| WORD | |
| WORDS | |
| WORKING | |
| WRITE | |

Extended BASIC's resident vocabulary is accessed by either the CALL SPGET subprogram or the CALL SAY subprogram. CALL SPGET is used to obtain the actual speech codes that are used by the speech synthesizer to produce a word. It has the following format:

CALL SPGET(word string,return string)

Word string can be a string constant, string variable, or string expression corresponding to one of the entries in the resident vocabulary. If a string constant is used, it must be enclosed in quotation marks. The subprogram returns the string of speech codes corresponding to word string and stores it in *return string*, which must be a string variable.

The following example demonstrates how CALL SPGET works. Notice that there are many blank spaces in the returned speech code string; that is because many of the codes cannot be printed.

```
100 CALL SPGET("HELLO",w$)
110 PRINT W$
```


The CALL SAY subprogram causes a word or words in the resident vocabulary to be spoken by the computer. If the subprogram is given a word which is not in the resident vocabulary, the computer will spell out the word rather than speak it. CALL SAY has the following format:

CALL SAY(word string,direct string,...)

Word string can be a string constant, string variable, or a string expression and should correspond to one or more of the entries in the resident vocabulary. If a string constant is used, it must be enclosed in quotation marks. *Direct string* is a string variable in which the speech codes corresponding to a word have been stored by the CALL SPGET subprogram. The CALL SAY subprogram may specify either of the two parameters exclusively, or it may specify a combination of the two. The two parameters are, however, positional; if one is omitted, its omission must be indicated by an additional comma.

The following examples demonstrate how CALL SAY works and illustrate the positional nature of its parameters.

```
100 CALL SAY("HELLO")
100 CALL SPGET("ARE",W$)
110 CALL SAY("WHO",W$,"YOU")
100 A$="HELLO"
110 B$="HOW ARE YOU"
120 CALL SAY(A$,B$)
100 CALL SPGET("GOODBYE",W$)
110 CALL SAY(,W$)
```

The speech produced by the CALL SAY subprogram can be made more natural by the use of commas (,), periods (.), and plus signs (+) within the specified string. As in actual speech, a comma indicates a slight pause, and a period indicates a slightly longer pause. A plus sign placed between words will cause them to sound more connected; in fact, the plus sign can be used to form new words out of old words (for example, hand + some = handsome). In addition to these symbols, number signs (#) may be used within a string to enclose phrases which are listed in the resident vocabulary as single entries (for example, #what was that#). If such phrases are not enclosed in number signs, the computer will spell out each word in the phrase.

The following examples demonstrate how these symbols may be used within the CALL SAY subprogram.

```
100 CALL SAY("HOW ARE YOU. WELL, ANSWER ME.")
100 CALL SAY("WHO ARE YOU")
110 FOR L=1 to 300 :: NEXT L
120 CALL SAY("WHO+ARE+YOU")
100 CALL SAY(I+AM+THE #TEXAS INSTRUMENTS#
HOME+COMPUTER")
```

Expanding the Resident Vocabulary

Although TI's resident vocabulary is limited to 373 entries, it is possible to expand the vocabulary by manipulating individual speech codes. Such manipulation lets you add suffixes to words or create new words from two or more old words. Suffixes can be added by concatenating the string of speech codes corresponding to the suffix with the string of speech codes corresponding to the word. New words can be created by concatenating part of the string of speech codes for one word with part of the string of speech codes for another word.

The CALL SPGET subprogram provides access to the speech codes for entries in the resident vocabulary. But to add a suffix, you need some way to obtain the speech codes for the suffix you have in mind. Fortunately, Texas Instruments shows you how in Appendix M of the Extended BASIC manual. That appendix lists seven subprograms which build speech code strings for the suffixes *-ing*, *-s*, *-ed*, and their variations (as determined by the type of word to which they are added). Since many of the speech codes cannot be printed, their corresponding ASCII codes can be used instead.

The speech-code string corresponding to a particular word ends with certain codes that affect the timing of sounds within the word to make it sound more natural. In order to add a suffix while keeping the natural sound, some of those trailing codes must be removed by truncating the speech-code string. Which codes to remove is best determined by trial and error. Truncation may also be used to isolate word segments for the formation of new words.

The following program, "Word Maker," demonstrates how truncation and concatenation are used to add suffixes to words and to create new words from old words.

How Word Maker Works

Line(s)

- 110-110 Clear the screen and display the STAND BY message.
- 120-350 Build the speech code strings for the seven suffixes. The speech codes for each suffix are stored as ASCII character codes in DATA statements at the end of the program. All the ASCII codes for a particular suffix are READ, converted to actual characters (CHR\$), and concatenated together. The concatenated strings are then stored in the subscripted variable SUF\$.
- 360-410 Clear the screen and present the main selection menu. The menu lets you choose between building a new word by adding a suffix or building a new word from two existing words.
- 420-670 Routine to build a new word by adding a suffix. Lines 440-480 display a menu which allows you to indicate the suffix you wish to add. Line 490 lets you enter the word to which the suffix will be added. Line 510 uses the CALL SPGET subprogram to obtain the speech string for the desired word. The length of the speech string is placed in the LW variable in line 520. Line 530 asks you to indicate how many bytes to truncate from the end of the word. This is a trial-and-error procedure; a good practice is to start with five or ten and increment by five. Line 540 makes sure you do not try to truncate more bytes than are contained in the string. Lines 550-570 build the new word. Line 550 subtracts truncated characters from the length of the word \$string. The -3 allows for the first three characters of the speech string, which are control characters. Line 560 builds the truncated word by concatenating the first two characters of the old string (control characters), the length of the new string (which is in the third position), and the remainder of the old string after truncation (determined by TR). Line 570 adds the appropriate suffix to the new string. Line 580 then causes the computer to "speak" the new word. Lines 590-600 allow you to try the word again with a different truncation number or go back to the main menu. Lines 610-660 display the word, suffix, and truncation number. These can be used in subsequent programs that will use the new word.
- 680-900 Build a new word by combining part of one word with another word. Lines 700-710 ask you to enter the two

words. Line 720 uses CALL SPGET to get the speech strings for the two words. Line 750 asks you the number of bytes to truncate from the first word. Lines 790-800 build the word by truncating the indicated number of bytes from the first word and then adding the second word to the end. Line 810 "speaks" the new word. Lines 820-830 let you try the word again or go back to the menu. Lines 840-890 display the two words and the truncation number for use by subsequent programs.

910-1100 DATA statments that contain the ASCII equivalents of the speech strings for the seven suffixes.

Note: When using Word Maker, do not try to truncate more than 50 bytes at a time, or the program may crash.

Program 6-1. Word Maker

```

100 CALL CLEAR
110 DISPLAY AT(10,9):"STAND BY..."
120 FOR L=1 TO 55
130 READ A :: SUF$(1)=SUF$(1)&CHR$(A)
140 NEXT L
150 FOR L=1 TO 29
160 READ A :: SUF$(2)=SUF$(2)&CHR$(A)
170 NEXT L
180 FOR L=1 TO 20
190 READ A :: SUF$(3)=SUF$(3)&CHR$(A)
200 NEXT L
210 FOR L=1 TO 37
220 READ A :: SUF$(4)=SUF$(4)&CHR$(A)
230 NEXT L
240 FOR L=1 TO 13
250 READ A :: SUF$(5)=SUF$(5)&CHR$(A)
260 NEXT L
270 FOR L=1 TO 29
280 READ A :: SUF$(6)=SUF$(6)&CHR$(A)
290 NEXT L
300 FOR L=1 TO 39
310 READ A :: SUF$(7)=SUF$(7)&CHR$(A)
320 NEXT L
330 FOR L=1 TO 7
340 READ A$ :: SUFFIX$(L)=A$
350 NEXT L
360 CALL CLEAR
370 DISPLAY AT(2,8):"WORDMAKER"
380 DISPLAY AT(6,1):"1 - ADD A SUFFIX" :: DISPLAY
    AT(8,1):"2 - MAKE A NEW WORD"

```

Speech Synthesis

```
390 DISPLAY AT(12,1):"SELECT----> _ " :: ACCEPT AT(
    12,13)VALIDATE("12")SIZE(-1)BEEP:OPT
400 ON OPT GOTO 420,680
410 GOTO 360
420 CALL CLEAR
430 DISPLAY AT(2,5):"ADD A SUFFIX"
440 DISPLAY AT(6,1):"1 - ADD ING" :: DISPLAY AT(8,
    1):"2 - ADD S AS IN CATS"
450 DISPLAY AT(10,1):"3 - ADD S AS IN CADS" :: DIS
    PLAY AT(12,1):"4 - ADD ES AS IN WISHES"
460 DISPLAY AT(14,1):"5 - ADD ED AS IN PASSED" ::
    DISPLAY AT(16,1):"6 - ADD ED AS IN CAUSED"
470 DISPLAY AT(18,1):"7 - ADD ED AS IN HEATED"
480 DISPLAY AT(20,1):"SELECT----> _ " :: ACCEPT AT(2
    0,12)VALIDATE("1234567")SIZE(-1)BEEP:SF
490 DISPLAY AT(22,1):"WHAT IS THE WORD?" :: ACCEPT
    AT(23,1)BEEP:WORD$
500 CALL CLEAR
510 CALL SPGET(WORD$,W$)
520 LW=LEN(W$)
530 DISPLAY AT(4,1):"TRUNCATE HOW MANY?" :: ACCEPT
    AT(4,20)VALIDATE(NUMERIC)SIZE(-2)BEEP:TRUN
540 IF TRUN>=LW-3 THEN 530
550 TR=LW-TRUN-3
560 NW$=SEG$(W$,1,2)&CHR$(TR)&SEG$(W$,4,TR)
570 NEWWORD$=NW$&SUF$(SF)
580 CALL SAY(,NEWWORD$)
590 DISPLAY AT(18,1):"NEW WORD OK? Y OR N _ " :: AC
    CEPT AT(18,21)VALIDATE("YN")SIZE(-1)BEEP:OK$
600 IF OK$="N" THEN 500
610 CALL CLEAR
620 DISPLAY AT(4,1):"WORD: ";WORD$
630 DISPLAY AT(6,1):"TRUNCATE: ";TRUN
640 DISPLAY AT(8,1):"SUFFIX: ";SUFFIX$(SF)
650 DISPLAY AT(12,2):"PRESS ANY KEY TO CONTINUE"
660 CALL KEY(3,K,S):: IF S=0 THEN 660
670 GOTO 360
680 CALL CLEAR
690 DISPLAY AT(2,5):"MAKE A NEW WORD"
700 DISPLAY AT(6,1):"FIRST WORD:" :: ACCEPT AT(6,1
    3)BEEP:FW$
710 DISPLAY AT(8,1):"SECOND WORD:" :: ACCEPT AT(8,
    14)BEEP:SW$
720 CALL SPGET(FW$,W$):: CALL SPGET(SW$,W2$)
730 CALL CLEAR
740 LW=LEN(W$)
750 DISPLAY AT(4,1):"TRUNCATE HOW MANY BYTES" :: D
    ISPLAY AT(5,1):"FROM FIRST WORD?"
760 ACCEPT AT(6,1)VALIDATE(NUMERIC)SIZE(-2)BEEP:TR
    UN
```

```

770 IF TRUN>LW-3 THEN 750
780 TR=LW-TRUN-3
790 NW$=SEG$(W$,1,2)&CHR$(TR)&SEG$(W$,4,TR)
800 NEWWORD$=NW$&W2$
810 CALL SAY(,NEWWORD$)
820 DISPLAY AT(18,1):"NEW WORD OK? Y OR N _" :: AC
    CEPT AT(18,21)VALIDATE("YN")SIZE(-1)BEEP:OK$
830 IF OK$="N" THEN 730
840 CALL CLEAR
850 DISPLAY AT(4,1):"1ST WORD: ";FW$
860 DISPLAY AT(6,1):"TRUNCATE: ";TRUN
870 DISPLAY AT(8,1):"2ND WORD: ";SW$
880 DISPLAY AT(12,2):"PRESS ANY KEY TO CONTINUE"
890 CALL KEY(3,K,S):: IF S=0 THEN 890
900 GOTO 360
910 REM ING SUFFIX
920 DATA 96,0,52,174,30,65,21,186,90,247,122,214,1
    79,95,77,13,202,50,153,120,117,57,40,248
930 DATA 133,173,209,25,39,85,225,54,75,167,29,77,
    105,91,44,157,118,180
940 DATA 169,97,161,117,218,25,119,184,227,222,249
    ,238,1
950 REM S SUFFIX (CATS)
960 DATA 96,0,26,14,56,130,204,0,223,177,26,224,10
    3,85,3,252,106,106,128,95,44,4,240,35,11,2,126
    ,16,121
970 REM S SUFFIX (CADS)
980 DATA 96,0,17,161,253,158,217,168,213,198,86,0,
    223,153,75,128,0,95,139,62
990 REM S SUFFIX (WISHES)
1000 DATA 96,0,34,173,233,33,84,12,242,205,166,55,
    173,93,222,68,197,188,134,238,123,102
1010 DATA 163,86,27,59,1,124,103,46,1,2,124,45,138
    ,129,7
1020 REM ED SUFFIX (PASSED)
1030 DATA 96,0,10,0,224,128,37,204,37,240,0,0,0
1040 REM ED SUFFIX (CAUSED)
1050 DATA 96,0,26,172,163,214,59,35,109,170,174,68
    ,21,22
1060 DATA 201,220,250,24,69,148,162,166,234,75,84,
    97,145,204,15
1070 REM ED SUFFIX (HEATED)
1080 DATA 96,0,36,173,233,33,84,12,242,205,166,183
    ,172,163,214,59,35,109,170,174,68,21
1090 DATA 22,201,92,250,24,69,148,162,38,235,75,84
    ,97,145,204,178,127
1100 DATA "ING","S AS IN CATS","S AS IN CADS","ES
    AS IN WISHES","ED AS IN PASSED","ED AS IN CAU
    SED",ED AS IN HEATED"

```

The following example will help demonstrate how the information obtained from Word Maker can be used in other programs. Suppose you were writing a program which included routines for loading and saving cassette files. Speech could be added to let the computer tell the user when the cassette recorder is rewinding the tape. The TI's resident vocabulary doesn't include the word *rewinding*, but it does include the word *rewind*. The *-ing* suffix can be added by using the Word Maker program to find the truncation value that sounds best. Once determined, that value (in this case, 32), along with the ASCII codes for the desired suffix, can be used to create the word *rewinding*. The process is illustrated in Program 6-2.

Program 6-2. Speech Demo 1

```
100 FOR L=1 TO 55
110 READ A :: ING$=ING$&CHR$(A)
120 NEXT L
130 CALL SPGET("REWIND",W$)
140 TR=LEN(W$)-32-3
150 NW$=SEG$(W$,1,2)&CHR$(TR)&SEG$(W$,4,TR)
160 NEWWORD$=NW$&ING$
170 CALL SAY("I AM",NEWWORD$,"THE CASSETTE.")
180 STOP
190 DATA 96,0,52,174,30,65,21,186,90,247,122,214,1
    79,95,77,13,202,50,153,120,117,57,40,248
200 DATA 133,173,209,25,39,85,225,54,75,167,29,77,
    105,91,44,157,118,180
210 DATA 169,97,161,117,218,25,119,184,227,222,249
    ,238,1
```

Taking this one step further, suppose you wanted the computer to remind the user to remove the cassette once that job was complete. The word *remove* is not in the resident vocabulary. However, the *re* portion of the word *read* can be concatenated with the word *move* to produce *remove*.

Word Maker would determine the truncation value (in this case, 50) needed to produce the *re* sound and use it to create the word *remove* in another program. This is illustrated in Program 6-3.

Program 6-3. Speech Demo 2

```
100 CALL SPGET("READ",W$)
110 CALL SPGET("MOVE",W2$)
120 TR=LEN(W$)-50-3
```

```

130 NW$=SEG$(W$,1,2)&CHR$(TR)&SEG$(W$,4,TR)
140 NEWWORD$=NW$&W2$
150 CALL SAY("NOW, YOU SHOULD",NEWWORD$,"THE CASSE
    TTE")
160 STOP

```

The Text-to-Speech Diskette

Texas Instruments' text-to-speech diskette is specifically designed to work with the Extended BASIC command module. It lets the computer speak almost any word in the English language and, unlike the Terminal Emulator II command module, enables this speech capability to be combined with the additional programming features of Extended BASIC. Obviously, a disk system (including a memory expansion unit, disk drive controller, and disk memory drive) is required.

To understand how this software works, it is helpful to know what phonemes and allophones are. A phoneme is the smallest unit of speech by which a difference in meaning can be perceived. For instance, the phonemes *b* and *c* distinguish the words *bat* and *cat*.

The same phoneme can represent slightly different sounds, depending on its position within a word and the letters that surround it. For instance, the *a* in *addition* has a slightly different sound than the *a* in *delta*. These different sounding forms of the same phoneme are called *allophones*.

Table 6-2 identifies the allophones used by the text-to-speech software, along with their numeric codes. The sound associated with each allophone is indicated by the way in which that allophone is used in a word.

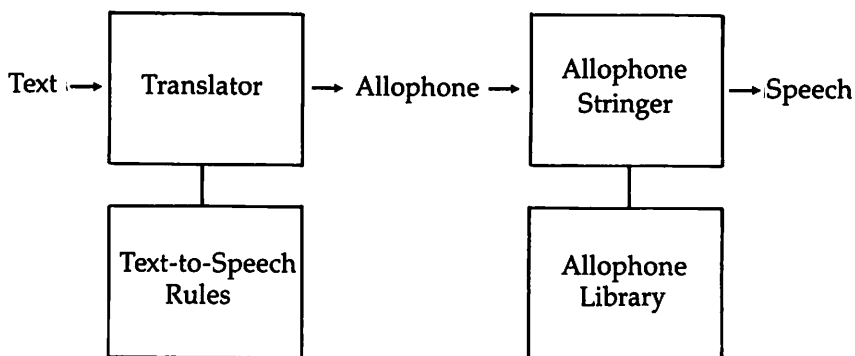
The text-to-speech software uses a specific set of speech rules to translate strings of text into strings of allophones. Those allophone strings are then converted to Linear Predictive Coding (LPC) strings and processed by the speech synthesizer to produce speech. A diagram illustrating this process is shown in Figure 6-1.

Table 6-2. Allophones

| Allophone Number | Sound | Allophone Number | Sound |
|---------------------|-------------------|---------------------|---------------|
| 1 | <i>addition</i> | 38 | <i>issue</i> |
| 2 | <i>annuity</i> | 39 | <i>choice</i> |
| 3 | <i>delta</i> | 40 | <i>cook</i> |
| 4 | <i>on time</i> | 41 | <i>poorly</i> |
| 5 | <i>autonomy</i> | 42 | <i>horse</i> |
| 6 | <i>anonymity</i> | 43 | <i>boat</i> |
| 7 | <i>eliminate</i> | 44 | <i>shoot</i> |
| 8 | <i>enough</i> | 45 | <i>hut</i> |
| 9 | <i>context</i> | 46 | <i>boot</i> |
| 10 | <i>ancient</i> | 47 | <i>had</i> |
| 11 | <i>western</i> | 48 | <i>odd</i> |
| 12 | <i>synthesis</i> | 49 | <i>hide</i> |
| 13 | <i>inane</i> | 50 | <i>card</i> |
| 14 | <i>took on</i> | 51 | <i>loud</i> |
| 15 | <i>donation</i> | 52 | <i>saw</i> |
| 16 | <i>annual</i> | 53 | <i>seed</i> |
| 17 | <i>unique</i> | 54 | <i>heel</i> |
| 18 | <i>above</i> | 55 | <i>hear</i> |
| 19 | <i>instrument</i> | 56 | <i>said</i> |
| 20 | <i>underneath</i> | 57 | <i>there</i> |
| 21 | <i>roses</i> | 58 | <i>day</i> |
| 22 | <i>basement</i> | 59 | <i>heard</i> |
| 23 | <i>seeker</i> | 60 | <i>hid</i> |
| 24 | <i>ratio</i> | 61 | <i>hill</i> |
| 25 | <i>funny</i> | 62 | <i>think</i> |
| 26 | <i>hat</i> | 63 | <i>boy</i> |
| 27 | <i>hot</i> | 64 | <i>could</i> |
| 28 | <i>height</i> | 65 | <i>poor</i> |
| 29 | <i>cart</i> | 66 | <i>core</i> |
| 30 | <i>house</i> | 67 | <i>low</i> |
| 31 | <i>sought</i> | 68 | <i>shoe</i> |
| 32 | <i>heat</i> | 69 | <i>mud</i> |
| 33 | <i>pierce</i> | 70 | <i>skull</i> |
| 34 | <i>set</i> | 71 | <i>pull</i> |
| 35 | <i>therapy</i> | 72 | <i>moon</i> |
| 36 | <i>take</i> | 73 | <i>like</i> |
| 37 | <i>hurt</i> | 74 | <i>bowl</i> |

| Allophone Number | Sound | Allophone Number | Sound |
|-----------------------------|--------------------|-----------------------------|--------------------|
| 75 | <i>awful, well</i> | 102 | <i>beige</i> |
| 76 | <i>may</i> | 103 | <i>skate</i> |
| 77 | <i>hum</i> | 104 | <i>case</i> |
| 78 | <i>nice</i> | 105 | <i>make</i> |
| 79 | <i>sane</i> | 106 | <i>key</i> |
| 80 | <i>think</i> | 107 | <i>cough</i> |
| 81 | <i>thing</i> | 108 | <i>space</i> |
| 82 | <i>real</i> | 109 | <i>pie</i> |
| 83 | <i>witch</i> | 110 | <i>nap</i> |
| 84 | <i>which</i> | 111 | <i>stake</i> |
| 85 | <i>you</i> | 112 | <i>tie</i> |
| 86 | <i>bad</i> | 113 | <i>late</i> |
| 87 | <i>dab</i> | 114 | <i>church</i> |
| 88 | <i>dig</i> | 115 | <i>fat</i> |
| 89 | <i>bid</i> | 116 | <i>laugh</i> |
| 90 | <i>give</i> | 117 | <i>hit</i> |
| 91 | <i>go</i> | 118 | <i>home</i> |
| 92 | <i>bag</i> | 119 | <i>hut</i> |
| 93 | <i>jug</i> | 120 | <i>seem</i> |
| 94 | <i>budge</i> | 121 | <i>miss</i> |
| 95 | <i>this</i> | 122 | <i>shine</i> |
| 96 | <i>clothe</i> | 123 | <i>wash</i> |
| 97 | <i>vine</i> | 124 | <i>thing</i> |
| 98 | <i>alive</i> | 125 | <i>with</i> |
| 99 | <i>zoo</i> | 126 | <i>short pause</i> |
| 100 | <i>does</i> | 127 | <i>long pause</i> |
| 101 | <i>azure</i> | | |

Figure 6-1. Text-to-Speech Diagram



The translation process is accomplished by three machine language routines—SETUP, XLAT, and SPEAK—which are accessed by Extended BASIC as subprograms. Before Extended BASIC can access these routines, however, the memory expansion unit must be initialized, and the three routines must be loaded into memory. In an Extended BASIC program, you can do it with the following lines; the text-to-speech diskette should be in Disk Drive 1.

```

100 CALL INIT
110 CALL LOAD("DSK1.SETUP","DSK1.XLAT",
    DSK1.SPEAK")
  
```

The CALL INIT subprogram prepares the computer to run machine language programs by making sure that the memory expansion unit is connected, removing any previously loaded subprograms from memory, and loading a set of supporting routines into memory. The CALL LOAD subprogram loads the machine language object files for the three routines.

Once loaded, the three routines can be accessed by Extended BASIC using the CALL LINK subprogram. CALL LINK passes control to a machine language subprogram.

The following CALL LINK statement causes the execution of the machine language routine SETUP.

```

120 CALL LINK("SETUP","DSK1.DATABASE")
  
```

SETUP loads the speech data base from the text-to-speech diskette into memory. This routine needs to be executed only once in a program; the speech data base will remain in memory until the memory expansion unit is turned off.

At that point the text-to-speech system is ready to convert text into speech. The XLAT routine is used to translate text strings into allophone strings, using a CALL LINK statement having the following format:

```
CALL LINK("XLAT",text string,allophone string)
```

Text string may be a string constant, string variable, or string expression, and represents the text string to be translated. If a string constant is used, it must be enclosed in quotation marks. Text string is limited to 128 characters; if this limit is exceeded, the error message STRING TRUNCATED is returned. XLAT returns the string of allophones corresponding to text string and stores it in *allophone string*, which must be a string variable. If the allophone string returned exceeds 255 characters, the error SPEECH STRING TOO LONG is returned.

Once the text string has been translated into an allophone string, the SPEAK routine is used to translate the allophone string into an LPC string and then into speech. The CALL LINK statement used to execute the "SPEAK" routine has the following format:

```
CALL LINK("SPEAK",allophone string,pitch,slope)
```

Allophone string may be a string variable or string expression and represents the allophone string to be translated into speech. *Pitch* is an integer in the range 0-63 and may be a numeric literal, numeric variable, or numeric expression. *Slope* is an integer in the range 0-255 and may be a numeric literal, numeric variable, or numeric expression. The normal values for the pitch and slope parameters are 43 and 128, respectively.

The pitch value determines the highness or lowness of the spoken sounds. A value of 1 produces the highest voice, while a value of 63 yields the lowest voice. If the value is 0, the resulting voice is much like a whisper. The frequency associated with each pitch value is given in Table 6-3.

Table 6-3. Pitch Frequencies

| Pitch | Freq. (Hz) | Pitch | Freq. (Hz) |
|-------|---------------|-------|---------------|
| 1 | 571.4 | 33 | 133.3 |
| 2 | 533.3 | 34 | 129.0 |
| 3 | 500.4 | 35 | 125.0 |
| 4 | 470.5 | 36 | 117.6 |
| 5 | 444.4 | 37 | 111.1 |
| 6 | 421.1 | 38 | 108.1 |
| 7 | 400.0 | 39 | 105.2 |
| 8 | 380.9 | 40 | 98.7 |
| 9 | 363.6 | 41 | 94.1 |
| 10 | 347.8 | 42 | 91.9 |
| 11 | 333.3 | 43 | 88.8 |
| 12 | 320.0 | 44 | 83.3 |
| 13 | 307.7 | 45 | 80.8 |
| 14 | 296.2 | 46 | 77.6 |
| 15 | 285.7 | 47 | 74.7 |
| 16 | 275.8 | 48 | 71.4 |
| 17 | 266.6 | 49 | 68.3 |
| 18 | 258.1 | 50 | 65.5 |
| 19 | 250.0 | 51 | 62.9 |
| 20 | 235.2 | 52 | 60.1 |
| 21 | 222.2 | 53 | 57.5 |
| 22 | 210.5 | 54 | 55.1 |
| 23 | 200.0 | 55 | 52.9 |
| 24 | 195.1 | 56 | 50.9 |
| 25 | 186.0 | 57 | 48.7 |
| 26 | 177.7 | 58 | 46.7 |
| 27 | 166.6 | 59 | 44.9 |
| 28 | 163.2 | 60 | 43.0 |
| 29 | 156.8 | 61 | 41.2 |
| 30 | 148.1 | 62 | 39.6 |
| 31 | 145.4 | 63 | 37.9 |
| 32 | 140.3 | | |

The slope value determines the rate at which the pitch changes within a spoken phrase. Best results are obtained when the slope is approximately 3.2 times the pitch. For example, if 40 is used for the pitch, the best value for the slope would be 3.2×40 , or 128. To prevent garbled speech, the slope should conform to the following limits:

slope < (pitch-1) * 16
slope < (63-pitch) * 16

The following program demonstrates how the text-to-speech software can be used by an Extended BASIC program to produce speech. It allows the user to enter any word or phrase (line 180), translates the word or phrase into an allophone string (line 200), and translates the allophone string into speech (line 210). In many cases, the words entered must be misspelled in order to produce the proper pronunciation.

In addition to words and phrases, sound effects can be created by entering various combinations of letters. Try the following combinations to get an idea of the sound effects that can be produced: TSTSTS, CKCKCK, YYYYYY, ZDZDZD, HAAALLP, and TKTKTK.

Program 6-4. Talker

```
100 CALL CLEAR
110 DISPLAY AT(6,10): "T A L K E R"
120 DISPLAY AT(14,10): "STAND BY..."
130 CALL INIT
140 CALL LOAD("DSK1.SPEAK", "DSK1.XLAT", "DSK1.SETUP
    ")
150 CALL LINK("SETUP", "DSK1.DATABASE")
160 CALL CLEAR
170 DISPLAY AT(4,1): "ENTER PHRASE TO BE SPOKEN:"
180 INPUT " ":PHR$
190 IF PHR$="" THEN 210
200 CALL LINK("XLAT",PHR$,ALL$)
210 CALL LINK("SPEAK",ALL$,43,128)
220 GOTO 160
```

In addition to translating alphabetic text, the text-to-speech routines also translate certain special characters. These special characters fall into four categories: numerical characters, pause-and-break characters, inflection symbols, and special symbols. The numerical characters consist of the digits 0 through 9 as well as the comma, decimal point, minus, and plus signs. When translated by the text-to-speech routines, the digits 0 through 9 will always be spoken. However, the other symbols in this category will be spoken only if they immediately precede a numerical digit.

The pause-and-break characters consist of the comma, period, exclamation point, question mark, colon, and semi-colon. The XLAT routine translates the comma into allophone number 126, which when translated by the SPEAK routine,

yields a short pause (50 milliseconds). The other pause-and-break characters are translated into allophone number 127 and produce a long pause (225 milliseconds). When used as pause-and-break characters, the comma and period must be followed by a space unless they appear as the last character in a text string. As will be seen, the pause-and-break characters also affect the inflection contour of a phrase or sentence which contains a primary stress symbol (one of the inflection symbols).

The inflection symbols consist of the primary stress symbol and the secondary stress symbol . The primary stress symbol indicates that primary emphasis should be given to the word it precedes. Only one primary stress symbol may be used in a phrase or sentence. If this primary stress is used in a phrase or sentence that ends with a comma or question mark, the phrase or sentence will be characterized by a progressive increase in pitch, starting at the primary stress symbol and ending at the comma or question mark. If the primary stress symbol is used in a phrase or sentence ended by any of the other pause-and-break characters, the phrase or sentence is characterized by a progressive decrease in pitch, starting at the primary stress symbol and ending at the pause-and-break character. When used in Program 6-4, "Talker," the following text strings will demonstrate the inflection contours (rising and falling) which can characterize a phrase or sentence.

"WHO ARE YOU?"

"WHO ARE YOU."

The primary stress symbol also indicates which syllable should be accented. If used alone, it indicates that the first syllable of the word is to be given primary emphasis. Used in conjunction with the shift indicator, it indicates that primary emphasis should be shifted to one of the other syllables in the word; the number of shift indicator symbols used determines which syllable. For example, allophone indicates that primary emphasis is to be given to the third syllable of the word *allophone*.

The secondary stress symbol indicates that secondary emphasis is to be given to the word it precedes. It causes the word it precedes to be spoken at a higher pitch. More than one secondary stress symbol may be used in a phrase or sentence, and each one encountered causes a progressive decline

in pitch (although all words preceded by a secondary stress symbol are spoken at a higher pitch than unstressed words). If a phrase or sentence contains secondary stress symbols, it should also contain a primary stress symbol; otherwise, the unstressed words will have a flat sound. The secondary stress symbol may also be used in conjunction with the shift indicator to shift secondary emphasis to other syllables in the word it precedes.

The special symbols category includes the symbols @, \$, %, &, *, (,), =, and /. When translated by the text-to-speech routines, these symbols are spoken as the words *at*, *dollar*, *percent*, *and*, *asterisk*, *open*, *close*, *equals*, and *slash*. The translation of special symbols and numerical characters is simply a matter of stringing together the allophones which make up the words corresponding to these characters. For example, the symbol @ is translated into the allophones corresponding to the word *at*.

Since pause-and-break characters and inflection symbols do not correspond to spoken words, they must be handled differently. They are translated with a number of inflection codes, which are listed in Table 6-4.

The following program will help demonstrate how text strings are translated into allophone strings. First, it allows the user to enter a text string. Then, after speaking the entered text, it lists the allophone numbers and inflection codes used to produce the corresponding speech.

Program 6-5. Allophone Number Lister

```

100 CALL CLEAR
110 DISPLAY AT(6,3):"ALLOPHONE NUMBER LISTER"
120 DISPLAY AT(14,10):"STAND BY..."
130 CALL INIT
140 CALL LOAD("DSK1.SETUP", "DSK1.XLAT", "DSK1.SPEAK
    ")
150 CALL LINK("SETUP", "DSK1.DATABASE")
160 CALL CLEAR
170 DISPLAY AT(4,1):"ENTER PHRASE:"
180 INPUT " ":PHR$
190 IF PHR$="" THEN 210
200 CALL LINK("XLAT", PHR$, ALL$)
210 CALL LINK("SPEAK", ALL$, 43, 128)
220 L=LEN(ALL$)
230 FOR X=1 TO L
240 PRINT ASC(SEG$(ALL$, X, 1))
250 NEXT X

```


Table 6-4. Inflection Codes

Code Number

- 249 Indicates a secondary stress symbol. In an allophone string, this code is placed before the first vowel allophone in the syllable which the stress symbol precedes.
- 250 Indicates a break in the text. It is placed at the beginning of an allophone string, as well as wherever a pause-and-break character appears. This code must be followed by two parameters. If a primary stress symbol is used in the text, the two parameters indicate the number of secondary stress symbols before and after the primary stress symbol. If the primary stress symbol is used in conjunction with a comma or question mark, the second parameter indicates the number of syllables after the primary stress symbol. The first parameter may optionally be given a value of either 254 or 255. A value of 254 indicates that the entire phrase which follows should have a rising contour; a value of 255 indicates that the entire phrase which follows should have a falling contour. When either of these values is used as the first parameter, the second parameter specifies the total number of syllables in the indicated phrase.
- 251 Indicates a new default slope value. It must be followed by one parameter indicating the new slope value to be used.
- 252 Indicates a new default pitch value. It must be followed by one parameter indicating the new pitch value to be used.
- 253 Indicates a primary stress symbol in a phrase characterized by a rising contour. In an allophone string, this code is placed before the first vowel allophone in the syllable which the stress symbol emphasizes.
- 254 Indicates a primary stress symbol in a phrase characterized by a falling contour. In an allophone string, this code is placed before the first vowel allophone in the syllable which the stress symbol emphasizes.
- 255 Indicates a temporary pitch level and modification for the allophone immediately following it. It must be followed by one parameter indicating the pitch value to be used.

Once you understand how to translate phrases and sentences, you can customize speech by stringing together the character equivalents of allophone numbers and inflection codes. The CHR\$ function is used to obtain the character equivalents; those characters are then concatenated together to form the allophone string used by the SPEAK routine. Here is how it can be done.

```
100 CALL INIT
110 CALL LOAD("DSK1.SETUP", "DSK1.XLAT", "DSK1.SPEAK
")
120 CALL LINK("SETUP", "DSK1.DATABASE")
* 130 W$=CHR$(252)&CHR$(30)&CHR$(53)&CHR$(49)&CHR$(2
52)&CHR$(33)&CHR$(53)&CHR$(49)&CHR$(252)&CHR$(
36)&CHR$(67)
140 CALL LINK("SPEAK", W$, 43, 128)
```

The following program actually lets you create custom speech. First, enter the number of syllables in the word or phrase to be created (line 170). Then, following the program prompts, enter the first allophone number or inflection code (line 210). The prompt will repeat until the number 300 is entered, indicating the final allophone number or inflection code. The resulting allophone string is then used by the SPEAK routine (line 260) to produce the corresponding speech.

Program 6-6. Allophone Stringer

```
100 CALL CLEAR
110 DISPLAY AT(6,5):"ALLOPHONE STRINGER"
120 DISPLAY AT(14,10):"STAND BY..."
130 CALL INIT
140 CALL LOAD("DSK1.SETUP", "DSK1.XLAT", "DSK1.SPEAK
")
150 CALL LINK("SETUP", "DSK1.DATABASE")
160 CALL CLEAR
170 DISPLAY AT(4,1):"ENTER NUMBER OF SYLLABLES" ::
    DISPLAY AT(6,1):"IN WORD:"
180 ACCEPT AT(6,10)VALIDATE(NUMERIC)BEEP:N
190 W$=CHR$(250)&CHR$(255)&CHR$(N)
200 CALL CLEAR
210 DISPLAY AT(10,1):"ENTER ALLOPHONE NUMBER" :: D
    ISPLAY AT(12,1):"OR 300 TO QUIT:"
```

Speech Synthesis

```
220 ACCEPT AT(12,17)VALIDATE(NUMERIC)BEEP:A
230 IF A=300 THEN 260
240 W$=W$&CHR$(A)
250 GOTO 200
260 CALL LINK("SPEAK",W$,43,128)
270 GOTO 160
```

Putting It All Together

Putting It All
Together

Graphics and sound are two of the most exciting features of home computers. Add them together, perhaps with a little speech synthesis thrown in, and you have programs with a great deal to offer. This chapter contains seven such programs. A complete explanation is included for each.

Mimic

This program illustrates that even modest use of graphics and sound can produce an enjoyable program. Its graphics consist of one horizontal bar, one vertical bar, and four solid-colored stationary sprites; its sound is limited to four distinct tones.

"Mimic" first puts a cross in the middle of the screen. A sequence of colored squares momentarily appears in the four quadrants, and with each color a single-frequency tone sounds. Its frequency depends on which quadrant contains the square. The object of the game is to repeat the sequence in which the squares appeared by pressing the appropriate number keys (1-4).

As many as six people can play at one time. On a player's first turn, only one square appears. Two squares appear on the second turn, and a new square is added after each correct response. An incorrect response ends the game.

How Mimic Works

Line(s)

- | | |
|---------|--|
| 100 | Seed the random number generator. |
| 110-140 | Define the patterns used by the program. ASCII code 91 defines the character used to draw the cross; code 96 is used to define the solid-colored sprites. The sprites may be one of four colors, depending on the quadrant in which they appear. Line 130 predefines the four colors. Line 140 clears the screen and sets it to light red. |
| 150-170 | Display the introduction banner, leave it on the screen for a few seconds, and then clear the screen. |

Putting It All Together

- 180–250 Determine the number of players, get each player's name, and set the difficulty level. The difficulty level is used to determine how long each square will be displayed on the screen.
- 260–280 Clear the screen and draw the cross.
- 290–350 Define the four sprites and display them in their appropriate quadrant on the screen until the players indicate they are ready to begin.
- 360 Set the color of all sprites (squares) to transparent (1). The sprites remain on the screen, but are invisible.
- 370–420 Initialize the work areas for the game. SEQ\$ contains the square sequence for a particular player. It is initially set to null. PLAYEROUT is a switch that tells the program if a particular player is still in the game. NUMOUT tells the program how many players are out of the game. When NUMOUT equals the number of players (PLAYERS), the game is over.
- 430–490 Begin the main game loop. Line 440 bypasses players that are out. Line 450 displays the current player's name, and lines 460–490 wait for him to indicate that he is ready for his turn.
- 500 Add a new square to a player's sequence. The new square is a random number between one and four. The string equivalent of the number is concatenated to the end of the player's previous square sequence (SEQ\$).
- 510–590 Display the sequence of squares, one at a time, on the screen. Line 510 goes through a loop the number of times indicated by the length of SEQ\$, which is equivalent to the number of squares. Line 520 determines the numeric value of each square stored in string format in SEQ\$. Lines 530–540 set on the color of the appropriate square, which makes it appear on the screen. Line 550 sounds the appropriate tone for that square. The duration is determined by the difficulty level. Line 560 invokes a time delay loop, also controlled by the difficulty level. This loop controls how long the square appears on the screen. Line 570 sets the square's color back to transparent, which effectively removes it from the screen.
- 600 Instructs the player to repeat the sequence of squares just presented.

- 610-860 Determine if the player responded with the correct sequence. Line 610 sets up the loop as determined by the number of squares. Lines 610-630 wait for a response from the player. Line 640 converts the ASCII code of the pressed key into its numeric value. Lines 650-690 turn on color for the square indicated by the player and sound the appropriate tone. Line 700 determines the actual square in the sequence. If it matches the key pressed by the player, line 710 branches to the end of the loop for that square. Otherwise, the key pressed by the player was incorrect. In this case, line 720 displays a message indicating an incorrect answer. Line 730 sounds a tone; lines 750-820 display the correct square on the screen; line 830 sets the switch indicating the player is out; and line 840 increments the number-of-players-out counter.
- 900-980 Display the tally for all players at the end of the game. The player with the highest tally is the winner.

Program 7-1. Mimic

```
100 RANDOMIZE
110 CALL CHAR(96,"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
   FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF")
120 CALL CHAR(91,"FFFFFFFFFFFFFF")
130 COLR(1)=9 :: COLR(2)=3 :: COLR(3)=14 :: COLR(4
   )=16
140 CALL CLEAR :: CALL SCREEN(10)
150 DISPLAY AT(8,10):"M I M I C"
160 FOR LOOP=1 TO 2000 :: NEXT LOOP
170 CALL CLEAR
180 DISPLAY AT(4,1):"NUMBER OF PLAYERS (1-6) _"
190 ACCEPT AT(4,25)VALIDATE("123456")SIZE(-1)BEEP:
   PLAYERS
200 FOR LOOP=1 TO PLAYERS
210 DISPLAY AT(6+(LOOP*2),1):"NAME OF PLAYER";LOOP
220 ACCEPT AT(6+(LOOP*2),18)BEEP:NAME$(LOOP)
230 NEXT LOOP
240 DISPLAY AT(22,1):"DIFFICULTY LEVEL (1-3)? _"
250 ACCEPT AT(22,25)VALIDATE("123")SIZE(-1)BEEP:LE
   VEL
260 CALL CLEAR :: CALL SCREEN(8)
270 CALL HCHAR(12,10,91,13)
280 CALL VCHAR(6,16,91,13)
290 CALL MAGNIFY(4)
```


Putting It All Together

```
300 CALL SPRITE(#1,96,9,47,78,#2,96,3,47,138,#3,96
,14,107,78,#4,96,16,107,138)
310 DISPLAY AT(9,10):"1";:: DISPLAY AT(9,18):"2";
320 DISPLAY AT(15,10):"3";:: DISPLAY AT(15,18):"4"
;
330 DISPLAY AT(22,1):"** PRESS ANY KEY TO BEGIN **"
"
340 CALL KEY(3,KEY,STATUS)
350 IF STATUS=0 THEN 340
360 CALL COLOR(#1,1,#2,1,#3,1,#4,1)
370 FOR LOOP=1 TO PLAYERS
380 PLAYEROUT(LOOP)=0
390 SEQ$(LOOP)=""
400 NEXT LOOP
410 NUMOUT=0
420 IF NUMOUT=PLAYERS THEN 890
430 FOR LOOP=1 TO PLAYERS
440 IF PLAYEROUT(LOOP)=1 THEN 870
450 DISPLAY AT(3,9):NAME$(LOOP);"S TURN"
460 DISPLAY AT(22,1):"---> PRESS ENTER WHEN READY"
470 CALL KEY(3,KEY,STATUS)
480 IF STATUS=0 THEN 470
490 DISPLAY AT(22,1):" "
500 SEQ$(LOOP)=SEQ$(LOOP)&STR$(1+INT(RND*4))
510 FOR SEQ=1 TO LEN(SEQ$(LOOP))
520 A=VAL(SEQ$(LOOP),SEQ,1))
530 CL=COLR(A)
540 CALL COLOR(#A,CL)
550 CALL SOUND(400*(4-LEVEL),300*A,0)
560 FOR DELAY=1 TO 100*(4-LEVEL):: NEXT DELAY
570 CALL COLOR(#A,1)
580 FOR DELAY=1 TO 20 :: NEXT DELAY
590 NEXT SEQ
600 DISPLAY AT(22,1):"** NOW REPEAT THE SEQUENCE"
610 FOR SEQ=1 TO LEN(SEQ$(LOOP))
620 CALL KEY(3,KEY,STATUS)
630 IF STATUS=0 OR KEY<49 OR KEY>52 THEN 620
640 B=VAL(CHR$(KEY))
650 CL=COLR(B)
660 CALL COLOR(#B,CL)
670 CALL SOUND(400*(4-LEVEL),300*B,0)
680 FOR DELAY=1 TO 100*(4-LEVEL):: NEXT DELAY
690 CALL COLOR(#B,1)
700 A=VAL(SEQ$(LOOP),SEQ,1))
710 IF A=B THEN 860
720 DISPLAY AT(22,2):"***** W R O N G *****"
730 CALL SOUND(1000,150,0,300,0)
740 FOR DELAY=1 TO 500 :: NEXT DELAY
750 CL=COLR(A)
```

```
760 FOR LOOP2=1 TO 10
770 CALL COLOR(#A,CL)
780 CALL SOUND(400,300*A,0)
790 FOR DELAY=1 TO 100 :: NEXT DELAY
800 CALL COLOR(#A,1)
810 FOR DELAY=1 TO 30 :: NEXT DELAY
820 NEXT LOOP2
830 PLAYEROUT(LOOP)=1
840 NUMOUT=NUMOUT+1
850 SEQ=9999
860 NEXT SEQ
870 NEXT LOOP
880 GOTO 420
890 CALL CLEAR
900 DISPLAY AT(4,1):"PLAYER" :: DISPLAY AT(4,21):"
    TURNS"
910 FOR LOOP=1 TO PLAYERS
920 DISPLAY AT(6+(2*LOOP),1):NAME$(LOOP)
930 DISPLAY AT(6+(2*LOOP),20):USING "#####":LEN(SE
    Q$(LOOP))-1
940 NEXT LOOP
950 DISPLAY AT(22,1):"PLAY AGAIN (Y/N)? _"
960 ACCEPT AT(22,19)VALIDATE("YN")SIZE(-1)BEEP:ANS
    $
970 IF ANS$="Y" THEN 260
980 CALL CLEAR :: STOP
```

Shooting Gallery

This program shows how moving sprites can be combined with sound to create a shooting gallery, complete with ducks, smiling faces, and diamonds for targets. Players use joysticks to control a crosshair sight, and the object is to position the sight over the targets and shoot them by pressing the fire button.

This game is designed for as many as four players. Each player has 15 shots, and there are a total of 13 targets. In addition, every turn must be completed within a specified time. If a player hits all 13 targets within the allotted time period, bonus points are awarded for each second that remains.

How Shooting Gallery Works

Line(s)

- | | |
|-----|--|
| 100 | Seed the random number generator. |
| 110 | DIMension the array variables used by the program. |

Putting It All Together

- 120-140 Set the point score for each of the three types of targets.
- 150-220 Define the character patterns used by the program. ASCII code 96 defines the ducks. ASCII code 97 defines the crosshair sight. ASCII code 100 defines the smiling faces. ASCII codes 98 and 102 define the diamonds, and ASCII code 101 defines the bars that separate the targets. Line 160 invokes the routine that displays the introduction banner.
- 230 Invoke the routines to display the instructions and determine the number of players and difficulty level.
- 240-290 Draw the shooting gallery background and tote board.
- 300-360 Define the sprites used in the program and set the targets in motion. The speed is determined by the difficulty level chosen.
- 370-400 Prompt the current player to get ready to start.
- 410-440 Set the initial values for the start of a player's turn. Each player has 15 shots (SH) and 30 seconds (TIME). TG is the counter for the number of targets hit. TI is the timer mechanism.
- 450-500 Main control loop. Line 450 checks for player response from the joystick and moves the crosshair accordingly. If the fire button was pressed, the shooting sound is made and program control is passed to line 510. Line 460 increments and displays the time. Line 480 decrements the shot counter and checks to see if all 15 shots have been taken or all targets hit.
- 510-680 Determine if a target was hit. Since there are 13 separate targets, checking all 13 sprites for coincidence would require coding 13 CALL COINC statements in a row. It was pointed out in Chapter 4 that this would probably cause the last five or six CALL COINC statements to miss a coincidence because of the slow speed of BASIC. Instead, line 510 determines the position of the crosshair sight. Then, using the row position of the sight, it determines which group of targets it could be over. The ducks are at the top of the screen. The smiling faces are in the center, and two groups of diamonds are at the bottom. The IF statement in line 510 branches to that particular group to check for coincidence. Lines 520-550 check the ducks; lines 560-600 check the smiling faces; lines 610-640 check the

- first group of diamonds; and lines 650–680 check the second group of diamonds.
- 690–710 End of player's turn.
- 720–810 End of game.
- 820–880 Award bonus points when all targets hit and time remains on clock.
- 890–970 Routine to indicate a hit. Lines 890–910 turn the target's color to red and then delete it. Lines 920–960 update the player's score.
- 980–1060 Introduction banner.
- 1070–1210 Instructions.
- 1220–1280 Determine number of players and difficulty level.

Program 7-2. Shooting Gallery

```

100 RANDOMIZE
110 DIM TOT(4),SC(14)
120 FOR L=2 TO 5 :: SC(L)=15 :: NEXT L
130 FOR L=6 TO 8 :: SC(L)=10 :: NEXT L
140 FOR L=9 TO 14 :: SC(L)=25 :: NEXT L
150 CALL CHAR(98,"081C3E7F7F3E1C08")
160 GOSUB 980
170 CALL CHAR(96,"0003021E3E7E0000")
180 CALL CHAR(97,"002020F820200000")
190 CALL CHAR(102,"10387CFEFE7C3810")
200 CALL CHAR(99,"FFFFFFFFFFFFFFFF")
210 CALL CHAR(101,"000000FFFF000000"):: CALL COLOR
    (9,7,1)
220 CALL CHAR(100,"FCB4FCCCFCB4CCFC")
230 GOSUB 1070 :: GOSUB 1220
240 CALL CLEAR :: CALL HCHAR(4,1,99,32):: CALL HCH
    AR(22,1,99,32)
250 PL=1
260 CALL HCHAR(9,1,101,32):: CALL HCHAR(15,1,101,3
    2)
270 DISPLAY AT(1,1):"PLAYER 1: ____ PLAYER 3: ____"
280 DISPLAY AT(3,1):"PLAYER 2: ____ PLAYER 4: ____"
290 P=1
300 CALL MAGNIFY(2)
310 CALL SPRITE(#2,96,12,40,20,0,S1,#3,96,12,54,85
    ,0,S1,#4,96,12,40,150,0,S1,#5,96,12,54,215,0,
    S1)
320 CALL SPRITE(#6,100,14,85,30,0,-S2,#7,100,14,85
    ,110,0,-S2,#8,100,14,85,190,0,-S2)

```

Putting It All Together

```
330 CALL SPRITE(#9,98,5,130,30,0,S3,#10,98,5,130,1
    10,0,S3,#11,98,5,130,190,0,S3)
340 CALL SPRITE(#12,102,5,155,30,0,-S3,#13,102,5,1
    55,110,0,-S3,#14,102,5,155,190,0,-S3)
350 CALL SPRITE(#1,97,2,90,120)
360 DISPLAY AT(23,1):" "
370 FOR L=1 TO 10
380 DISPLAY AT(23,11):"PLAYER";PL :: CALL SOUND(30
    ,600,4):: FOR L2=1 TO 20 :: NEXT L2
390 DISPLAY AT(23,11):"{12 SPACES}" :: FOR L2=1 TO
    15 :: NEXT L2
400 NEXT L
410 SH=15 :: TG=0
420 DISPLAY AT(24,1):"SHOTS LEFT:" :: CALL HCHAR(2
    4,16,43,SH)
430 DISPLAY AT(23,1):"TIME:"
440 TI=0 :: TIME=30
450 CALL JOYST(1,X,Y):: CALL MOTION(#1,-Y*2,X*2)::
    CALL KEY(1,K,S):: IF S<>0 THEN CALL SOUND(60,
    -6,0):: GOTO 510
460 TI=TI+1 :: TIME=30-INT(TI/5):: DISPLAY AT(23,7
    ):TIME :: IF TIME=0 THEN 690
470 GOTO 450
480 SH=SH-1 :: IF SH=0 OR TG=13 THEN 690
490 DISPLAY AT(24,1):"SHOTS LEFT:" :: CALL HCHAR(2
    4,16,43,SH)
500 GOTO 450
510 CALL POSITION(#1,DR,DC):: IF DR<70 THEN 560 EL
    SE IF DR>140 THEN 650 ELSE IF DR>110 THEN 610
520 CALL COINC(#1,#6,7,C):: IF C=-1 THEN SP=6 :: G
    OTO 890
530 CALL COINC(#1,#7,7,C):: IF C=-1 THEN SP=7 :: G
    OTO 890
540 CALL COINC(#1,#8,7,C):: IF C=-1 THEN SP=8 :: G
    OTO 890
550 GOTO 480
560 CALL COINC(#1,#2,7,C):: IF C=-1 THEN SP=2 :: G
    OTO 890
570 CALL COINC(#1,#3,7,C):: IF C=-1 THEN SP=3 :: G
    OTO 890
580 CALL COINC(#1,#4,7,C):: IF C=-1 THEN SP=4 :: G
    OTO 890
590 CALL COINC(#1,#5,7,C):: IF C=-1 THEN SP=5 :: G
    OTO 890
600 GOTO 480
610 CALL COINC(#1,#9,TL,C):: IF C=-1 THEN SP=9 ::
    GOTO 890
620 CALL COINC(#1,#10,TL+1,C):: IF C=-1 THEN SP=10
    :: GOTO 890
```

Putting It All Together

```
630 CALL COINC(#1,#11,TL+1,C):: IF C=-1 THEN SP=11
    :: GOTO 890
640 GOTO 480
650 CALL COINC(#1,#12,TL+1,C):: IF C=-1 THEN SP=12
    :: GOTO 890
660 CALL COINC(#1,#13,TL+2,C):: IF C=-1 THEN SP=13
    :: GOTO 890
670 CALL COINC(#1,#14,TL+2,C):: IF C=-1 THEN SP=14
    :: GOTO 890
680 GOTO 480
690 DISPLAY AT(24,1):"[5 SPACES]GAME OVER PLAYER";
    PL :: CALL SOUND(900,800,5):: FOR L=1 TO 800 :
        : NEXT L :: DISPLAY AT(24,1):" "
700 IF TG=13 AND TIME>0 THEN GOSUB 820
710 IF PL<NP THEN PL=PL+1 :: GOTO 290
720 FOR L=1 TO 4
730 DISPLAY AT(24,1):" " :: CALL SOUND(40,400,4)
740 FOR L2=1 TO 20 :: NEXT L2
750 DISPLAY AT(24,7):"PLAY AGAIN Y/N?"
760 FOR L2=1 TO 15 :: NEXT L2
770 NEXT L
780 CALL KEY(3,K,S):: IF S=0 THEN 780
790 IF K=78 THEN CALL CLEAR :: STOP
800 IF K=89 THEN TOT(1)=0 :: TOT(2)=0 :: TOT(3)=0
    :: TOT(4)=0 :: CALL DELSPRITE(ALL):: GOSUB 122
    0 :: GOTO 240
810 GOTO 720
820 CALL SOUND(700,200,5,900,5)
830 DISPLAY AT(24,1):"*** BONUS POINTS AWARDED ***"
    "
840 FOR L=1 TO 800 :: NEXT L
850 TOT(PL)=TOT(PL)+(15*TIME)
860 DISPLAY AT(ROW,COL):TOT(PL);
870 DISPLAY AT(24,1):" "
880 RETURN
890 CALL SOUND(50,400,0)
900 CALL COLOR(#SP,9):: FOR L=1 TO 5 :: NEXT L
910 CALL DELSPRITE(#SP)
920 TG=TG+1
930 TOT(PL)=TOT(PL)+SC(SP)
940 IF PL/2=INT(PL/2)THEN ROW=3 ELSE ROW=1
950 IF PL=1 OR PL=2 THEN COL=10 ELSE COL=25
960 DISPLAY AT(ROW,COL):TOT(PL);
970 GOTO 480
980 CALL CLEAR :: CALL SCREEN(10)
990 DISPLAY AT(5,13):"T I" :: DISPLAY AT(8,7):"S H
    O O T I N G" :: DISPLAY AT(11,8):"G A L L E R
    Y"
1000 FOR L=1 TO 1000 :: NEXT L
```

Putting It All Together

```
1010 FOR L=1 TO 10
1020 CALL HCHAR(INT(RND*22+1),INT(RND*26+1),98,1):
      : CALL SOUND(50,-6,0)
1030 FOR L2=1 TO 20 :: NEXT L2
1040 NEXT L
1050 FOR L=1 TO 500 :: NEXT L
1060 RETURN
1070 CALL CLEAR :: CALL SCREEN(8)
1080 DISPLAY AT(4,1):"DO YOU NEED INSTRUCTIONS?" :
      : DISPLAY AT(6,1):"RESPOND Y/N"
1090 CALL SOUND(100,400,4)
1100 CALL KEY(3,K,S):: IF S=0 THEN 1100
1110 IF K=78 THEN 1210
1120 IF K<>89 THEN 1080
1130 CALL CLEAR
1140 CALL SPRITE(#2,96,12,15,20,#3,100,14,38,20,#4
      ,98,5,63,20):: CALL MAGNIFY(2)
1150 DISPLAY AT(3,8):"15 POINTS" :: DISPLAY AT(6,8
      ):"10 POINTS" :: DISPLAY AT(9,8):"25 POINTS"
1160 DISPLAY AT(12,1):"USE THE JOYSTICK TO POSITIO
      N" :: DISPLAY AT(14,1):"THE CROSSHAIR ON THE
      TARGET"
1170 DISPLAY AT(16,1):"AND PRESS THE FIRE BUTTON T
      O" :: DISPLAY AT(18,1):"SHOOT. UP TO 4 PEOP
      LE CAN"
1180 DISPLAY AT(20,1):"PLAY AT ONE TIME. EACH" ::
      DISPLAY AT(22,1):"PLAYER HAS 15 SHOTS."
1190 DISPLAY AT(24,7):"PRESS ANY KEY"
1200 CALL KEY(3,K,S):: IF S=0 THEN 1200
1210 CALL DELSPRITE(ALL):: RETURN
1220 CALL CLEAR
1230 DISPLAY AT(4,1):"NUMBER OF PLAYERS? _" :: ACC
      EPT AT(4,20)VALIDATE("1234")SIZE(-1)BEEP:NP
1240 DISPLAY AT(8,1):"TARGET SPEED? (1-3) _" :: DI
      SPLAY AT(10,1):"(3 = FASTEST)"
1250 ACCEPT AT(8,21)VALIDATE("123")SIZE(-1)BEEP:DL
1260 S1=3*DL :: S2=2*DL :: S3=5*DL
1270 TL=4+(2*DL)
1280 RETURN
```

Alphabet Invasion

Take a challenging mind teaser, throw in some graphics and sound, and you quickly have an enjoyable program for your home computer. "Alphabet Invasion" is such a program.

The object of this game is to unscramble letters beamed down by the alien spaceship. Up to four players may compete,

and each player has five words to unscramble. Scoring is based on speed. If you let the timer run down to zero, your turn is over.

How Alphabet Invasion Works

Line(s)

- | | |
|---------|--|
| 100-110 | Clear the screen and seed the random number generator. |
| 120 | Set the dimension limit for the program's vocabulary. Up to 200 words can be used in the program at one time. |
| 130-150 | Read the words stored in DATA statements into the array. The program will continue to read DATA statements until the word END is encountered. |
| 160-220 | Define the character patterns used in the program. ASCII code 96 defines the alien spaceship; ASCII code 97 defines the beam from the ship; ASCII code 104 is a blank character used to erase the beam; ASCII codes 112, 113, and 114 are used to vaporize the letters; and ASCII code 120 is used as a horizontal separation character. |
| 230-520 | Present the introduction banner. Line 270 displays the name of the game. Lines 290-330 define the spaceship sprite and start it moving from left to right at the top of the screen. Lines 340-360 shoot a beam toward the word ALPHABET, and lines 370-400 "vaporize" it. Lines 420-440 and 450-480 do the same thing to the word INVASION. Lines 500-520 move the spaceship off the screen. |
| 540-570 | Determine the number of players and the name of each. |
| 580-630 | Display the game screen. |
| 640-690 | Main game loop. There will be five words to unscramble for each player. Line 660 indicates the current player. |
| 700-800 | Select a word from the array. Line 700 selects a random number between 1 and NW, which is the number of words in the array. If that element in the array is a null, which means it has already been used, another random number is selected. Line 710 puts the selected word in WORD\$. Line 730 sets the variable NL to the length of the selected word and clears the variable |

Putting It All Together

- SCRAMBLE\$. Line 740 clears the array used to scramble the word. Lines 750–780 scramble the word by randomly selecting numbers between 1 and NL. Line 760 insures that each number is used only once. Line 770 uses the selected number to place that position of the word in the next available position of SCRAMBLE\$. Line 790 makes sure the scrambled word isn't the same as the original word.
- 810–960 Present the scrambled word to the player. Lines 810–840 move the spaceship across the top of the screen. Lines 850–900 beam down each letter in the scrambled word. Lines 910–950 move the ship off the screen.
- 970–1040 Control loop for player's response. Line 990 looks for keyboard input. Line 1000 checks to see if the timer has run down to zero. If it has, the turn is over and the program branches to line 1050. Line 1010 increments the counter and displays it on the screen. Line 1020 checks to see if the ENTER key was pressed. The ENTER key is pressed after the player has guessed the word, so the program branches to line 1070. Line 1030 checks to see if the back arrow (FNCT S) key was pressed. This allows the player to correct an entry. Line 1040 handles any other key. The ASCII code for the key is converted to a character and displayed on the screen.
- 1050–1060 Indicate to the player that time has run out.
- 1070–1180 Check to see if the guess is correct. Line 1070 displays RIGHT or WRONG accordingly. Lines 1090–1110 vaporize the scrambled letters. Line 1120 erases the scrambled letters and displays the correct word. Lines 1150–1170 calculate the player's score and display it on the screen.
- 1190–1200 Let player use back arrow key (FNCT S) to correct a letter.
- 1210–1270 End of game.
- 1280–1410 DATA statements containing words. You may add your own words to this list or replace it altogether. The maximum number is 200 and the last word must be END.

Program 7-3. Alphabet Invasion

```

100 CALL CLEAR
110 RANDOMIZE
120 DIM WORDS$(200)
130 NW=NW+1
140 READ WORDS$(NW)
150 IF WORDS$(NW) <> "END" THEN 130 ELSE NW=NW-1
160 CALL CHAR(96,"3C7EFFFF7E3C0000")
170 CALL CHAR(97,"1818181818181818")
180 CALL CHAR(104,"0000000000000000")
190 CALL CHAR(112,"10234719F618C291")
200 CALL CHAR(113,"048385718496F719")
210 CALL CHAR(114,"748D7391F3831174")
220 CALL CHAR(120,"0000FFFFFFFF0000")
230 CALL SCREEN(2):: CALL COLOR(9,9,1,10,1,1,11,11
,1,12,14,1)
240 FOR L=2 TO 8
250 CALL COLOR(L,16,1)
260 NEXT L
270 DISPLAY AT(15,7):"A L P H A B E T" :: DISPLAY
AT(21,7):"I N V A S I O N"
280 FOR L=1 TO 1000 :: NEXT L
290 CALL MAGNIFY(2)
300 CALL SPRITE(#1,96,5,20,1)
310 FOR L=2 TO 120
320 CALL LOCATE(#1,20,L):: CALL SOUND(-1,500-L,7,-
3,0)
330 NEXT L
340 CALL VCHAR(4,16,97,12):: CALL SOUND(20,1500,5,
-6,0)
350 CALL VCHAR(4,16,104,12)
360 CALL SOUND(150,110,9,-6,0)
370 FOR L=1 TO 4
380 FOR L2=112 TO 114
390 CALL HCHAR(15,9,L2,15)
400 NEXT L2 :: NEXT L
410 CALL HCHAR(15,7,104,17)
420 CALL VCHAR(4,16,97,17):: CALL SOUND(20,1500,5,
-6,0)
430 CALL VCHAR(4,16,104,17)
440 CALL SOUND(150,110,9,-6,0)
450 FOR L=1 TO 4
460 FOR L2=112 TO 114
470 CALL HCHAR(21,9,L2,15)
480 NEXT L2 :: NEXT L
490 CALL HCHAR(21,7,104,17)
500 FOR L=121 TO 256
510 CALL LOCATE(#1,20,L):: CALL SOUND(-1,380+L,7,-
3,0)

```

Putting It All Together

```
520 NEXT L :: CALL DELSPRITE(#1)
530 CALL CLEAR
540 DISPLAY AT(4,1):"NUMBER OF PLAYERS? (1-4) _" :
    : ACCEPT AT(4,26)VALIDATE("1234")SIZE(-1)BEEP:NP
550 FOR L=1 TO NP
560 DISPLAY AT(6+(L*2),1):"NAME OF PLAYER";L :: AC
    CEPT AT(6+(L*2),18)SIZE(9)BEEP:PN$(L)
570 NEXT L
580 CALL CLEAR
590 CALL HCHAR(16,1,120,32):: CALL HCHAR(19,1,120,
    32)
600 DISPLAY AT(18,1):"TIME:" :: DISPLAY AT(18,12):
    "WORD: "
610 DISPLAY AT(20,9):"** SCORE **"
620 DISPLAY AT(22,1):PN$(1);:: DISPLAY AT(22,16):P
    N$(2);
630 DISPLAY AT(23,1):PN$(3);:: DISPLAY AT(23,16):P
    N$(4);
640 FOR PLAY=1 TO 5
650 FOR L=1 TO NP
660 DISPLAY AT(1,8):"TURN: ";PN$(L)
670 GOSUB 700 :: GOSUB 810 :: GOSUB 970
680 NEXT L
690 NEXT PLAY :: GOTO 1210
700 WORD=1+INT(RND*NW):: IF WORDS$(WORD)="" THEN 7
    00
710 WORD$=WORDS$(WORD)
720 WORDS$(WORD)=""
730 NL=LEN(WORD$):: SCRAMBLE$=""
740 FOR L2=1 TO NL :: WA(L2)=0 :: NEXT L2
750 FOR L2=1 TO NL
760 N=1+INT(RND*NL):: IF WA(N)=1 THEN 760
770 SCRAMBLE$=SCRAMBLE$&SEG$(WORD$,N,1):: WA(N)=1
780 NEXT L2
790 IF SCRAMBLE$=WORD$ THEN 730
800 RETURN
810 CALL SPRITE(#1,96,5,20,1)
820 FOR L2=2 TO 84
830 CALL LOCATE(#1,20,L2):: CALL SOUND(-1,500,7,-3
    ,0)
840 NEXT L2
850 FOR L2=1 TO NL
860 CALL VCHAR(4,10+(L2*2),97,9):: CALL SOUND(20,1
    500,5,-6,0):: CALL VCHAR(4,10+(L2*2),104,9)
870 CALL HCHAR(14,10+(L2*2),ASC(SEG$(SCRAMBLE$,L2,
    1)),1)
880 FOR L3=1 TO 16
890 CALL LOCATE(#1,20,68+(L2*16)+L3):: CALL SOUND(
    -1,380,7,-3,0)
```

```

900 NEXT L3 :: NEXT L2
910 CALL POSITION(#1,DR,DC)
920 FOR L2=DC TO 256
930 CALL LOCATE(#1,20,L2):: CALL SOUND(-1,500,7,-3
,0)
940 NEXT L2
950 CALL DELSPRITE(#1)
960 RETURN
970 TI=0 :: TIME=100 :: ANS$=""
980 DISPLAY AT(18,19):" _"
990 CALL KEY(3,K,S)
1000 IF TIME=0 THEN 1050
1010 TI=TI+1 :: TIME=100-INT(TI/8):: DISPLAY AT(18
,6):TIME:: IF S=0 THEN 990
1020 IF K=13 THEN 1070
1030 IF K=8 THEN 1190
1040 ANS$=ANS$&CHR$(K):: DISPLAY AT(18,19):ANS$::
GOTO 990
1050 CALL SOUND(1000,800,5,-6,0)
1060 DISPLAY AT(12,6):"*** OUT OF TIME ***" :: GOTO
1090
1070 IF ANS$=WORD$ THEN DISPLAY AT(12,9):"*** RIGHT
***" ELSE DISPLAY AT(12,9):"*** WRONG ***" :: C
ALL SOUND(1000,800,5,-6,0):: TIME=0 :: GOTO 1
090
1080 CALL SOUND(1000,200,4,2000,0)
1090 FOR L3=1 TO 8 :: FOR L4=112 TO 114
1100 CALL HCHAR(14,12,L4,LEN(WORD$)*2-1)
1110 NEXT L4 :: NEXT L3
1120 CALL HCHAR(14,1,32,32):: CALL HCHAR(15,1,32,3
2):: DISPLAY AT(15,12):WORD$ :: FOR DELAY=1 T
O 2000 :: NEXT DELAY
1130 CALL HCHAR(12,1,32,32):: CALL HCHAR(14,1,32,3
2):: CALL HCHAR(15,1,32,32)
1140 DISPLAY AT(18,6):"100":: DISPLAY AT(18,19):"
"
1150 SC(L)=SC(L)+TIME
1160 R=22+INT(L/3):: C=10 :: IF L=2 OR L=4 THEN C=
C+15
1170 DISPLAY AT(R,C):USING "###":SC(L);
1180 RETURN
1190 IF LEN(ANS$)<2 THEN ANS$="" ELSE ANS$=SEG$(AN
S$,1,LEN(ANS$)-1)
1200 DISPLAY AT(18,19):ANS$ :: GOTO 990
1210 CALL HCHAR(1,1,32,32)
1220 DISPLAY AT(1,8):"*** GAME OVER ***"
1230 CALL SOUND(500,220,0,294,0,349,0)
1240 CALL SOUND(500,196,0,330,0,698,0)
1250 CALL SOUND(500,220,0,294,0,349,0)

```

Putting It All Together

```
1260 CALL SOUND(900,247,0,587,0,784,0)
1270 GOTO 1270
1280 DATA "TRAIN", "BOOKS", "DESK", "PAPER", "SOCKS", "
      BOUNCE", "WRITE", "CLOUT"
1290 DATA "COMPUTE", "BORDER", "THINK", "NUMERIC", "RE
      GRET", "FUNGUS", "MONITOR", "PRINTER", "TONSIL"
1300 DATA "RULER", "PURSUE", "ANIMAL", "FEMALE", "CLOS
      ET", "CABLE", "CURTAIN", "TOWEL"
1310 DATA "SAUCER", "INVADE", "PICKLE", "CURDLE", "STR
      EET", "AVENUE", "PANTS", "GRAPHIC", "KNIGHT", "PER
      SON", "RADICAL"
1320 DATA "GIGGLE", "LEATHER", "MEDIUM", "OBLONG", "SQ
      UARE", "JACKET", "BREATH", "AUTHOR", "AUTUMN", "TR
      OWEL", "METHOD"
1330 DATA "PISTOL", "ROUGH", "ROTUND", "CLENCH", "SURM
      ISE", "TRANSIT", "TRAMPLE", "VALIANT", "WEIGHT", "
      HEIGHT", "YEOMAN"
1340 DATA "ZIPPER", "ZEALOUS", "PROJECT", "JUNGLE", "H
      EAVY", "HEDGE", "JUDGE", "QUICK", "CONSIST", "BELI
      EVE", "BISCUIT", "HARMONY", "MUSICAL", "SCALE"
1350 DATA "RADIO", "VALUE", "ARRAY", "CHILD", "MONTH",
      "CROWN", "SHELL", "BIRTH", "PROGRAM", "STRING", "A
      SSIGN", "TRACK"
1360 DATA "ADDRESS", "POINT", "ANALOGY", "VERSION", "E
      NOUGH", "COMPANY", "REASON", "GRADE", "WORLD", "CH
      AIR", "GUIDE", "QUENCH", "TURMOIL"
1370 "STATION", "MAGIC", "RESOLVE", "MACHINE", "MANAGE
      R", "VITAL", "OBSCURE", "REVOLT", "EDUCATE", "PROD
      UCE", "ELECT"
1380 DATA "DRIVE", "ENTRY", "ELICIT", "ASPIRE", "REFUT
      E", "BANANA", "OUTSIDE", "NORMAL", "MAJOR", "ACCEN
      T", "TRASH", "BINDER", "SPIRAL", "REMOTE"
1390 DATA "SEARCH", "CAREER", "BEACH", "LISTING", "JOI
      NT", "CAVITY", "WRECK", "MANUAL", "ECONOMY", "EXPA
      ND", "REVULSE", "DESTROY", "REPAIR"
1400 DATA "CAPTIVE", "SOUND", "WAYWARD", "CRAFT", "IMP
      ULSE", "IMAGINE", "CRACKER", "BECOME", "BEMOAN", "
      AVOID", "FETCH", "FUTURE", "INJURY", "JUSTICE"
1410 DATA "END"
```

Banzai Bunny

"Banzai Bunny" demonstrates just how versatile TI sprites can be. It uses 24 of the 28 possible sprites and incorporates many sprite subprogram commands, including CALL MAGNIFY, CALL MOTION, CALL POSITION, and CALL DELSPRITE. In addition, it uses two versions of the CALL COINC subprogram.

Your job is to move the bunny from the bottom of the screen across six lanes of traffic, over a polluted river on the backs of turtles, and finally to safety on the far shore. The bunny is controlled by the E, S, and D keys, and points are scored by reaching the other side of the river. As many as nine players may compete at one time. Each player has five bunnies, and a player's turn continues until all five have been lost.

How Banzai Bunny Works

Line(s)

- 100 Clear the screen.
- 110–220 Define the character patterns used in the program. ASCII code 96 defines the bunny moving up, ASCII code 100 defines the bunny moving to the right, and ASCII code 104 defines the bunny moving to the left. ASCII codes 108, 112, and 116 define the bunny jumping up, right, and left, respectively. ASCII code 140 defines the bunny's demise on the highway or in the river. ASCII codes 120 and 124 defines the cars, while ASCII code 128 defines the turtles.
- 230 Set the character pattern colors.
- 240–250 Determine the number of players.
- 260–450 Set up the screen. Lines 270–350 draw the highway, the river, and the borders. Lines 370–420 define the car sprites and set them in motion. Lines 430–450 define the turtle sprites and set them in motion.
- 460–510 Main game loop. Line 460 initiates the loop based on the number of players. Line 470 sounds a series of two tones to indicate the beginning of a player's turn. Lines 480–490 display the player number, number of bunnies, and score. Line 500 transfers program control to the routine to control the bunny.
- 520 Branch to end-of-game routine when no more players.
- 530 Define the bunny sprite.
- 540–600 Loop to control the bunny while on the road. Line 540 looks for keyboard input. Line 550 transfers to a different routine if the bunny is at a row position less than 64, which indicates that the bunny is at the river. Otherwise, the bunny is on the road. Line 560 checks to see if there is any coincidence. If there is, it can only mean that the bunny was run over by a car. In that case, the program branches to line 780. Lines 570–590

Putting It All Together

- check to see if an arrow key was pressed. If so, the routine to move the bunny in that direction is invoked.
- 610 Control routine for the bunny while at the river. Line 610 checks for coincidence. If there is, the bunny is on a turtle, and consequently OK. If there is no coincidence, the bunny is in the river. This is the opposite condition from the routine for the road.
- 620 Sound routine for jumping bunny.
- 640-690 Routines to move the bunny. Lines 640-650 move the bunny up; lines 660-670 move the bunny to the left; lines 680-690 move the bunny to the right.
- 700-770 Loop to control bunny at the river. Line 700 looks for keyboard input. Line 710 invokes the appropriate routine to move the bunny. Line 720 determines the bunny's position. If the bunny is at row 1, he has reached the other side of the river. Line 730 makes sure that the bunny does not wrap around the screen horizontally. If the bunny reaches the edge of the screen, he is dead. Based on the row position obtained in line 720, line 740 puts the bunny in motion to the left or right, at the speed of the turtle at that row position. The effect is to have the rabbit ride the turtle. Line 750 checks for coincidence. If there is no coincidence, then the rabbit is in the river. In that case, the program branches to the routine at line 780.
- 780-840 Routine for a bunny's demise. Line 780 produces a squish sound. Line 790 changes the bunny's pattern. Line 800 is a delay loop. Line 810 deletes the bunny sprite. Line 820 decrements the bunny counter. If there are no more bunnies for the player, the program branches back to the main routine for the next player.
- 850-880 Routine to increment and display score for a bunny that made it to the other side of the river.
- 890-940 End of game. The final score for all players is displayed.

Program 7-4. Banzai Bunny

```
100 CALL CLEAR :: RANDOMIZE
110 CALL CHAR(40, "FFFFFFFFFFFFFFFF")
120 CALL CHAR(96, "0000000030709030303030100000000
0000000040C0808080C0C0C0000000000")
```

Putting It All Together

```
130 CALL CHAR(100,"0000000000000070F0F0E0000000000000
00000000002010D8F8E03000000000000000")
140 CALL CHAR(104,"00000000004081B1F070C0000000000000
00000000000000E0F0F07000000000000000")
150 CALL CHAR(108,"000000003070903030303030100000000
0000004040408080808080C040400000")
160 CALL CHAR(112,"0000000000000070F0F38000000000000
00000000002010D8F8E01C00000000000000")
170 CALL CHAR(116,"00000000004081B1F073C000000000000
00000000000000E0F0F01C00000000000000")
180 CALL CHAR(120,"00000000000102043F7FFFFFF3810000
000000000000F04844FEFFFFFFF1C080000")
190 CALL CHAR(124,"0000000000000F12227FFFFFF3810000
000000000000804020FCFEFFFFFF1C080000")
200 CALL CHAR(128,"0000070F3F3F7FFFFFF7F3F3F0F07000
00000E0F0FCFCFEFFFFFFEFCFCF0E00000")
210 CALL CHAR(132,"FFFFFFFFFFFFFFFF"):: CALL CHAR(
136,"0000000000000000F")
220 CALL CHAR(140,"80C0607030300806030303060718180
0010203071C30708080C0607030100100")
230 CALL COLOR(2,5,1,13,4,1,14,2,1)
240 DISPLAY AT(4,1):"NUMBER OF PLAYERS? "
250 ACCEPT AT(4,20)VALIDATE(NUMERIC)SIZE(-1)BEEP:N
P
260 CALL CLEAR
270 CALL HCHAR(23,1,132,32):: CALL HCHAR(24,1,132,
32)
280 CALL HCHAR(10,1,132,32):: CALL HCHAR(9,1,132,3
2)
290 CALL HCHAR(1,1,132,32):: CALL HCHAR(2,1,132,32
)
300 FOR L=12 TO 21 STEP 2
310 CALL HCHAR(L,1,136,32)
320 NEXT L
330 FOR L=3 TO 8
340 CALL HCHAR(L,1,40,32)
350 NEXT L
360 CALL MAGNIFY(3)
370 CALL SPRITE(#2,124,2,160,70,0,9,#3,124,9,160,1
50,0,9,#4,124,14,160,230,0,9)
380 CALL SPRITE(#5,124,15,144,10,0,12,#6,124,5,144
,100,0,12,#24,124,13,144,190,0,12)
390 CALL SPRITE(#7,124,2,128,70,0,10,#8,124,9,128,
150,0,10,#9,124,11,128,230,0,10)
400 CALL SPRITE(#10,120,9,112,70,0,-10,#11,120,14,
112,150,0,-10,#12,120,16,112,230,0,-10)
410 CALL SPRITE(#13,120,11,96,10,0,-12,#14,120,5,9
6,100,0,-12,#25,120,14,96,190,0,-12)
```


Putting It All Together

```
420 CALL SPRITE(#15,120,2,80,70,0,-9,#16,120,10,80
,150,0,-9,#17,120,15,80,230,0,-9):: GOTO 460
430 CALL SPRITE(#19,128,2,48,10,0,4)
440 CALL SPRITE(#21,128,2,32,10,0,-3)
450 CALL SPRITE(#22,128,2,16,10,0,5):: RETURN
460 FOR PL=1 TO NP
470 BUNNY=5 :: FOR D=1 TO 5 :: CALL SOUND(300,500,
0):: CALL SOUND(300,300,0):: NEXT D
480 DISPLAY AT(24,1):USING "##### #":"PLAYER",PL;
:: DISPLAY AT(24,20):USINC "##### #":"BUNNIE
S",BUNNY
490 DISPLAY AT(1,10):"SCORE:{4 SPACES}";
500 GOSUB 530
510 NEXT PL
520 GOTO 890
530 GOSUB 430 :: CALL SPRITE(#18,96,16,176,125)::
R=176 :: C=125
540 CALL KEY(3,K,S)
550 IF R<64 THEN G10
560 CALL COINC(ALL,CO):: IF CO<>0 THEN 780
570 IF K=69 THEN GOSUB 640 :: IF R<64 THEN 700 ELS
E 540
580 IF K=83 THEN GOSUB 660 :: GOTO 540
590 IF K=68 THEN GOSUB 680 :: GOTO 540
600 GOTO 540
610 CALL COINC(ALL,CO):: IF CO=0 THEN 780 ELSE 570
620 CALL SOUND(70,500,5,-6,0):: CALL SOUND(30,300,
5,-4,0):: RETURN
630 RETURN
640 CALL PATTERN(#18,108):: R=R-16 :: SW=1 :: IF R
<1 THEN R=1
650 CALL LOCATE(#18,R,C):: CALL PATTERN(#18,96)::
GOSUB 620 :: RETURN
660 CALL PATTERN(#18,116):: C=C-16 :: SW=1 :: IF C
<1 THEN C=1
670 CALL LOCATE(#18,R,C):: CALL PATTERN(#18,104)::
GOSUB 620 :: RETURN
680 CALL PATTERN(#18,112):: C=C+16 :: SW=1 :: IF C
>240 THEN C=240
690 CALL LOCATE(#18,R,C):: CALL PATTERN(#18,100)::
GOSUB 620 :: RETURN
700 CALL KEY(3,K,S)
710 IF S<>0 THEN IF K=69 THEN GOSUB 640 ELSE IF K=
83 THEN GOSUP 660 ELSE IF K=68 THEN GOSUB 680
720 CALL POSITION(#18,R,C):: IF R=1 THEN CALL MOTI
ON(#18,0,0):: GOTO 850
730 IF C<7 OR C>238 THEN 780
740 IF R<=48 AND R>32 THEN CALL MOTION(#18,0,4)ELS
E IF R<=32 AND R>16 THEN CALL MOTION(#18,0,-3)
ELSE CALL MOTION(#18,0,5)
```

```
750 IF SW=1 THEN CALL COINC(ALL,CO):: IF CO=0 THEN
    780 ELSE SW=0
770 GOTO 700
780 CALL SOUND(300,3500,0,5000,5):: CALL SOUND(-20
    0,200,5,-6,0)
790 CALL PATTERN(#18,140):: CALL COLOR(#18,11)
800 FOR D=1 TO 25 :: NEXT D
810 CALL DELSPRITE(#18)
820 BUNNY=BUNNY-1 :: IF BUNNY<1 THEN 630
830 DISPLAY AT(24,27):USING "###":BUNNY
840 GOTO 530
850 CALL SOUND(1500,400,5,1000,0):: FOR D=1 TO 500
    :: NEXT D
860 SC(PL)=SC(PL)+15
870 DISPLAY AT(1,17):USING "###":SC(PL);
880 GOTO 530
890 CALL DELSPRITE(ALL):: CALL CLEAR
900 DISPLAY AT(2,7):"** SCORE **"
910 FOR L=1 TO NP
920 DISPLAY AT(4+L*2,4):USING "##### #{3 SPACES}#
    ##": "PLAYER",L,SC(L)
930 NEXT L
940 GOTO 940
```

Zone Defender

While the previous programs have used the keyboard or joystick to move a sprite around the screen, "Zone Defender" demonstrates how sprites can be used to simulate a moving window. In other words, your laser sight will remain stationary while everything else moves.

You are the commander of a defensive space station at the outer reaches of the solar system, and you are responsible for defending your zone from invading spacecraft. The screen is your window into space, and at its center is a targeting sight. When an invading craft flies across the screen, you must track it with your sight and destroy it. The joysticks control your maneuvering rockets, which rotate the ship in the appropriate direction.

The maneuvering rockets have a cumulative rotation effect. In other words, the longer the joysticks are pushed in a certain direction, the faster the rotation of your ship. The ship will continue to rotate in one direction until an equal amount of rocket power is applied in the other direction.

To destroy an invading craft, you must first center it in your sight. A message at the top of the screen will tell you

when you're locked on, and you can fire your laser using the fire button on your joystick. There are ten invading ships, and your final score is determined by how quickly you dispatch the invaders.

How Zone Defender Works

Line(s)

- 100 Clear the screen and set its color to black.
- 110 Seed the random number generator.
- 120-200 Define the character patterns used by the program. ASCII codes 100, 104, and 108 define the stars. ASCII code 96 defines the sight. ASCII code 112 defines the enemy spacecraft, ASCII code 120 defines the laser, and ASCII codes 124 and 128 define the explosion characters used when a ship is destroyed.
- 210 Set the character code colors.
- 220-270 Set the magnification factor to 3. Create the sprites for the stars.
- 280 Transfer to the introduction banner.
- 290 Display the score.
- 300 Create the sprite for the sight.
- 310-320 Increment the enemy ship counter. If count exceeds ten, branch to the end-of-game routine. Otherwise, display the number of enemy ships remaining.
- 330-350 Determine a random row and column velocity for the enemy spacecraft. Line 350 places the spacecraft on the screen.
- 360 Check for joystick movement. If there is movement, produces a sound to simulate rockets firing.
- 370 Check to see if the spacecraft is in the sight. If it is, display the LOCKED ON message.
- 380 Check to see if fire button was pressed. If it was, branch to laser firing routine at line 440.
- 390-400 Put stars in motion, based on the direction in which the joystick is moved. Effect is cumulative.
- 410-420 Control the spacecraft's direction and speed based on the direction in which the joystick is pressed. Again, the effect is cumulative.
- 430 Check the joystick again.

- | | |
|---------|---|
| 440-530 | Fire the laser. Line 470 checks to see if the sight and the spacecraft were coincident. If they weren't, the program branches back to the main routine. Otherwise, the spacecraft was hit. Lines 480-500 generate an exploding sound and change the spacecraft sprite to the explosion patterns. Line 510 displays the score. Line 520 stops all motion in preparation for the next spacecraft. |
| 540-550 | End of game. |
| 560-600 | Display introduction banner. |

Program 7-5. Zone Defender

[illegible]

Putting It All Together

```
330 X2=1+INT(RND*30):: IF RND<.5 THEN X2=-X2
340 Y2=1+INT(RND*30):: IF RND<.5 THEN Y2=-Y2
350 CALL SPRITE(#3,112,15,1+INT(RND*190),1+INT(RND
    *240),X2,Y2)
360 CALL JOYST(1,A,B):: TI=TI+1 :: IF A<>0 OR B<>0
    THEN CALL SOUND(-540,-6,0)
370 CALL COINC(#1,#3,6,CO):: IF CO<>0 THEN DISPLAY
    AT(2,10):"LOCKED ON";ELSE DISPLAY AT(2,10):"
"
380 CALL KEY(1,K,S):: IF K=18 THEN 440
390 Y=MIN(Y+B,120):: Y=MAX(Y,-120):: X=MIN(X-A,120
):: X=MAX(X,-120)
400 CALL MOTION(#4,Y,X,#5,Y,X,#6,Y,X,#7,Y,X,#8,Y,X
    ,#9,Y,X,#10,Y,X)
410 Y2=MIN(Y2-B,120):: Y2=MAX(Y2,-120):: X2=MIN(X2
    -A,120):: X2=MAX(X2,-120)
420 CALL MOTION(#3,-Y2,X2)
430 GOTO 360
440 CALL SPRITE(#2,120,9,90,125)
450 CALL SOUND(100,1200,4,-5,0):: CALL SOUND(-200,
    900,5,-5,0)
460 CALL DELSPRITE(#2)
470 IF CO=0 THEN GOTO 360
480 CALL COLOR(#3,9):: CALL PATTERN(#3,124):: CALL
    SOUND(-200,400,0,-6,0):: SC=SC+MAX(500-TI,0):
    : TI=0
490 CALL PATTERN(#3,128)
500 CALL DELSPRITE(#3)
510 DISPLAY AT(24,9):USING "####":SC
520 CALL MOTION(#4,0,0,#5,0,0,#6,0,0,#7,0,0,#8,0,0
    ,#9,0,0,#10,0,0):: X=0 :: Y=0
530 GOTO 310
540 DISPLAY AT(2,8):"*** GAME OVER ***"
550 GOTO 550
560 DISPLAY AT(6,11):"Z O N E"
570 DISPLAY AT(10,7):"D E F E N D E R"
580 DISPLAY AT(20,8):"PRESS ANY KEY"
590 CALL KEY(3,K,S):: IF S=0 THEN 590
600 CALL CLEAR :: RETURN
```

Addition Climber

Since small children are usually unafraid of computers, it is little wonder that home computers are used more and more frequently in education.

A computer program designed for educational purposes should have several important characteristics. First, it must allow for increasing levels of difficulty in order to challenge

the child. Second, it should establish some specific goal as a measure of success. Finally, it should offer encouragement and a reward when that goal is achieved.

"Addition Climber" allows your child to practice addition skills. The program uses the TI Speech Synthesizer to offer encouragement or inform the child of an incorrect answer. Speech synthesis gives any educational program a more personal approach.

The program asks for the answers to ten addition problems. For each correct answer, a gorilla climbs a little further up a building. If all ten answers are correct, the gorilla reaches the top of the building and does a short dance.

How Addition Climber Works

Line(s)

| | |
|---------|---|
| 100 | Clear the screen and seed the random number generator. |
| 110-180 | Define the character patterns used in the program. |
| 190 | Set the colors for the character patterns. |
| 200 | Determine the difficulty level. The highest number used in the problems is determined by the difficulty level times five. In other words, the highest numbers to add for a difficulty level four would be $20 + 20$. |
| 210-310 | Set up the problem board. Line 230 draws the building; line 270 creates the gorilla. |
| 320-490 | Main game loop. Line 340 determines the two numbers for each addition problem. Line 350 invokes a routine that presents the number on the screen as magnified sprites. Lines 360-390 "speak" the problem question (for example, "What is 10 plus 12?"). Line 400 accepts the answer from the child. Line 410 determines if the answer is correct and invokes the appropriate response routine. Line 420 calls a routine to display the correct answer on the screen as 1 or 2 magnified sprites. Line 430-450 branch to routines which speak words of encouragement. Line 460 displays the tally of correct answers. Line 480 deletes the number sprites in preparation for the next problem. |
| 500-540 | End of game. If all problems were answered correctly, line 500 calls routines to speak congratulations to the child and make the gorilla dance. |
| 550-600 | Control which number is spoken. |

Putting It All Together

- 610-680 Indicate an incorrect answer. Line 640 randomly picks one of three phrases to inform the child the answer is wrong. Lines 650-670 inform the child of the correct answer.
- 690-710 The three phrases for an incorrect answer.
- 720-800 Routine for a correct answer. Again, one of three phrases is randomly picked to inform the child that the answer is correct.
- 810-830 The three phrases for a correct answer.
- 840-990 Display the numbers for the current problem as magnified sprites.
- 1000-1070 Display the correct answer as magnified sprites.
- 1080-1160 Words of encouragement.
- 1170-1220 Make the gorilla climb the building.
- 1230-1320 Make the gorilla dance at the top of the building when all answers are correct.
- 1330-2030 Speak the numbers.

Program 7-6. Addition Climber

```
100 CALL CLEAR :: RANDOMIZE
110 CALL CHAR(40, "000000FFFF000000")
120 CALL CHAR(96, "FFFFFFFFFFFFFFFF")
130 CALL CHAR(116, "0101010101010101")
140 CALL CHAR(120, "FFF1FF81FF81FF81")
150 CALL CHAR(104, "FFFC3C3C3C3FFFF")
160 CALL CHAR(112, "9999FF3C3C7E4242")
170 CALL CHAR(113, "1818FF3C3C3C2424")
180 CALL CHAR(114, "1818FFBDBD3C2424")
190 CALL COLOR(9,13,1,10,15,5,11,2,1,12,9,16)
200 DISPLAY AT(6,1): "DIFFICULTY LEVEL 1-7 " :: AC
    CEPT AT(6,22)VALIDATE("1234567")SIZE(-1)BEEP:D
    L
210 CALL CLEAR
220 FOR L=19 TO 23 :: CALL HCHAR(L,1,96,14):: NEXT
    L
230 FOR L=8 TO 18 :: CALL HCHAR(L,5,104,5):: NEXT
    L
240 CALL VCHAR(4,7,116,4):: CALL HCHAR(4,8,120,1)
250 DISPLAY AT(1,5): "ADDITION CLIMBER"
260 CALL MAGNIFY(2)
270 CALL SPRITE(#10,112,2,128,40):: ROW=128 :: COL
    =40
```

Putting It All Together

```
280 DISPLAY AT(4,18):"TRIES:" :: DISPLAY AT(6,18):  
    "RIGHT:"  
290 CALL HCHAR(18,23,40,4)  
300 CALL SAY("PRESS ANY KEY TO START.")  
310 CALL KEY(3,K,S):: IF S=0 THEN 310  
320 FOR L=1 TO 10  
330 DISPLAY AT(8,10):" " :: DISPLAY AT(23,13):" "  
340 V1=1+INT(RND*(DL*5)): V2=1+INT(RND*(DL*5))  
350 GOSUB 840  
360 CALL SAY("WHAT IS"):: IF V1>25 THEN A1=V1-25 E  
    LSE A=V1  
370 IF V1>25 THEN GOSUB 570 ELSE GOSUB 550  
380 CALL SAY("AND"):: IF V2>25 THEN A1=V2-25 ELSE  
    A=V2  
390 IF V2>25 THEN GOSUB 570 ELSE GOSUB 550  
400 DISPLAY AT(23,13):"ANSWER-->" :: ACCEPT AT(23,  
    23)VALIDATE(NUMERIC)BEEP:ANS$  
410 IF VAL(ANS$)<>V1+V2 THEN GOSUB 610 ELSE GOSUB  
    720  
420 GOSUB 1000  
430 IF L=5 AND RIGHT=5 THEN GOSUB 1080  
440 IF L=7 AND RIGHT=7 THEN GOSUB 1100  
450 IF L=9 AND RIGHT=9 THEN GOSUB 1120  
460 DISPLAY AT(4,25):USING "##":L :: DISPLAY AT(6,  
    25):USING "##":RIGHT  
470 FOR D=1 TO 1700 :: NEXT D  
480 CALL DELSPRITE(#1,#2,#3,#4,#5,#6,#7)  
490 NEXT L  
500 IF RIGHT=10 THEN GOSUB 1230 :: GOSUB 1140  
510 CALL SAY("DO YOU WANT TO TRY AGAIN"):: CALL SA  
    Y("IF SO, PRESS THE Y KEY")  
520 CALL KEY(3,K,S):: IF S=0 THEN 520  
530 IF K=89 THEN CALL DELSPRITE(ALL):: RUN  
540 CALL DELSPRITE(ALL):: CALL CLEAR :: STOP  
550 ON A GOSUB 1340,1350,1360,1370,1380,1390,1400,  
    1410,1420,1430,1440,1450,1460,1470,1480,1490,1  
    500,1510,1520,1530,1540,1550,1560,1570,1580  
  
560 RETURN  
570 ON A1 GOSUB 1590,1600,1610,1620,1630,1640,1650  
    ,1660,1670,1680,1690,1700,1710,1720,1730,1740,  
    1750,1760,1770,1780,1790,1800,1810,1820,1830  
580 RETURN  
590 ON A1 GOSUB 1840,1850,1860,1870,1880,1890,1900  
    ,1910,1920,1930,1940,1950,1960,1970,1980,1990,  
    2000,2010,2020,2030  
600 RETURN  
610 A=V1+V2  
620 DISPLAY AT(8,18):"*** WRONG ***"
```


Putting It All Together

```
630 IF A>50 THEN A1=A-50 ELSE IF A>25 THEN A1=A-25
640 ON 1+INT(RND*3)GOSUB 690,700,710
650 CALL SAY("THE CORRECT ANSWER IS")
660 IF A>50 THEN GOSUB 590 ELSE IF A>25 THEN GOSUB
    570
670 IF A<=25 THEN GOSUB 550
680 RETURN
690 CALL SAY("UHOH. THAT IS NOT THE RIGHT ANSWER."
):: RETURN
700 CALL SAY("SORRY, BUT THAT IS NOT RIGHT."):: RE
    TURN
710 CALL SAY("NO, THAT IS NOT RIGHT."):: RETURN
720 A=V1+V2 :: IF A>50 THEN A1=A-50 ELSE IF A>25 T
    HEN A1=A-25
730 RIGHT=RIGHT+1
740 DISPLAY AT(8,18): "*** RIGHT ***"
750 ON 1+INT(RND*3)GOSUB 810,820,830
760 CALL SAY("THE CORRECT ANSWER IS")
770 IF A>50 THEN GOSUB 590 ELSE IF A>25 THEN GOSUB
    570
780 IF A<=25 THEN GOSUB 550
790 GOSUB 1170
800 RETURN
810 CALL SAY("VERY GOOD. YOU GOT IT RIGHT."):: RE
    TURN
820 CALL SAY("THAT IS RIGHT"):: RETURN
830 CALL SAY("THAT IS EXACTLY RIGHT."):: RETURN
840 S1=0 :: S2=0
850 FOR SL=10 TO V1 STEP 10
860 S1=S1+1
870 NEXT SL
880 S2=V1-(S1*10):: SC2=48+S2
890 IF S1=0 THEN SC1=32 ELSE SC1=48+S1
900 CALL SPRITE(#1,SC1,2,80,176,#2,SC2,2,80,192)
910 S1=0 :: S2=0
920 FOR SL=10 TO V2 STEP 10
930 S1=S1+1
940 NEXT SL
950 S2=V2-(S1*10):: SC2=48+S2
960 IF S1=0 THEN SC1=32 ELSE SC1=48+S1
970 CALL SPRITE(#3,SC1,2,112,176,#4,SC2,2,112,192)
980 CALL SPRITE(#5,43,2,112,160)
990 RETURN
1000 S1=0 :: S2=0
1010 FOR SL=10 TO V1+V2 STEP 10
1020 S1=S1+1
1030 NEXT SL
1040 S2=(V1+V2)-(S1*10):: SC2=48+S2
1050 IF S1=0 THEN SC1=32 ELSE SC1=48+S1
```

Putting It All Together

```
1060 CALL SPRITE(#6,SC1,2,144,176,#7,SC2,2,144,192
)
1070 RETURN
1080 CALL SAY("YOU ARE DOING VERY WELL.")
1090 RETURN
1100 CALL SAY("YOU ARE DOING VERY GOOD WORK.")
1110 RETURN
1120 CALL SAY("YOU HAVE NINE RIGHT. ONLY ONE MORE
TO GO.")
1130 RETURN
1140 CALL SAY("YOU GOT ALL TEN PROBLEMS RIGHT. YO
U MUST BE A COMPUTER.")
1150 FOR D=1 TO 700 :: NEXT D
1160 RETURN
1170 ROW=ROW-8
1180 CALL PATTERN(#10,113)
1190 CALL LOCATE(#10,ROW,COL)
1200 FOR D=1 TO 10 :: NEXT D
1210 CALL PATTERN(#10,112)
1220 RETURN
1230 ROW=ROW-8 :: CALL LOCATE(#10,ROW,COL)
1240 FOR L=1 TO 18
1250 FOR L2=0 TO 1
1260 CALL PATTERN(#10,112+(L2*2))
1270 CALL LOCATE(#10,ROW-(L2*8),COL)
1280 NEXT L2
1290 FOR D=1 TO 40 :: NEXT D
1300 NEXT L
1310 CALL LOCATE(#10,ROW,COL)
1320 RETURN
1330 CALL SAY("ZERO"):: RETURN
1340 CALL SAY("ONE"):: RETURN
1350 CALL SAY("TWO"):: RETURN
1360 CALL SAY("THREE"):: RETURN
1370 CALL SAY("FOUR"):: RETURN
1380 CALL SAY("FIVE"):: RETURN
1390 CALL SAY("SIX"):: RETURN
1400 CALL SAY("SEVEN"):: RETURN
1410 CALL SAY("EIGHT"):: RETURN
1420 CALL SAY("NINE"):: RETURN
1430 CALL SAY("TEN"):: RETURN
1440 CALL SAY("ELEVEN"):: RETURN
1450 CALL SAY("TWELVE"):: RETURN
1460 CALL SAY("THIRTEEN"):: RETURN
1470 CALL SAY("FOURTEEN"):: RETURN
1480 CALL SAY("FIFTEEN"):: RETURN
1490 CALL SAY("SIX TEEN"):: RETURN
1500 CALL SAY("SEVEN TEEN"):: RETURN
1510 CALL SAY("EIGHT TEEN"):: RETURN
```

Putting It All Together

1520 CALL SAY("NINE TEEN"):: RETURN
1530 CALL SAY("TWENTY"):: RETURN
1540 CALL SAY("TWENTY ONE"):: RETURN
1550 CALL SAY("TWENTY TWO"):: RETURN
1560 CALL SAY("TWENTY THREE"):: RETURN
1570 CALL SAY("TWENTY FOUR"):: RETURN
1580 CALL SAY("TWENTY FIVE"):: RETURN
1590 CALL SAY("TWENTY SIX"):: RETURN
1600 CALL SAY("TWENTY SEVEN"):: RETURN
1610 CALL SAY("TWENTY EIGHT"):: RETURN
1620 CALL SAY("TWENTY NINE"):: RETURN
1630 CALL SAY("THIRTY"):: RETURN
1640 CALL SAY("THIRTY ONE"):: RETURN
1650 CALL SAY("THIRTY TWO"):: RETURN
1660 CALL SAY("THIRTY THREE"):: RETURN
1670 CALL SAY("THIRTY FOUR"):: RETURN
1680 CALL SAY("THIRTY FIVE"):: RETURN
1690 CALL SAY("THIRTY SIX"):: RETURN
1700 CALL SAY("THIRTY SEVEN"):: RETURN
1710 CALL SAY("THIRTY EIGHT"):: RETURN
1720 CALL SAY("THIRTY NINE"):: RETURN
1730 CALL SAY("FORTY"):: RETURN
1740 CALL SAY("FORTY ONE"):: RETURN
1750 CALL SAY("FORTY TWO"):: RETURN
1760 CALL SAY("FORTY THREE"):: RETURN
1770 CALL SAY("FORTY FOUR"):: RETURN
1780 CALL SAY("FORTY FIVE"):: RETURN
1790 CALL SAY("FORTY SIX"):: RETURN
1800 CALL SAY("FORTY SEVEN"):: RETURN
1810 CALL SAY("FORTY EIGHT"):: RETURN
1820 CALL SAY("FORTY NINE"):: RETURN
1830 CALL SAY("FIFTY"):: RETURN
1840 CALL SAY("FIFTY ONE"):: RETURN
1850 CALL SAY("FIFTY TWO"):: RETURN
1860 CALL SAY("FIFTY THREE"):: RETURN
1870 CALL SAY("FIFTY FOUR"):: RETURN
1880 CALL SAY("FIFTY FIVE"):: RETURN
1890 CALL SAY("FIFTY SIX"):: RETURN
1900 CALL SAY("FIFTY SEVEN"):: RETURN
1910 CALL SAY("FIFTY EIGHT"):: RETURN
1920 CALL SAY("FIFTY NINE"):: RETURN
1930 CALL SAY("SIXTY"):: RETURN
1940 CALL SAY("SIXTY ONE"):: RETURN
1950 CALL SAY("SIXTY TWO"):: RETURN
1960 CALL SAY("SIXTY THREE"):: RETURN
1970 CALL SAY("SIXTY FOUR"):: RETURN
1980 CALL SAY("SIXTY FIVE"):: RETURN
1990 CALL SAY("SIXTY SIX"):: RETURN

```
2000 CALL SAY("SIXTY SEVEN"):: RETURN
2010 CALL SAY("SIXTY EIGHT"):: RETURN
2020 CALL SAY("SIXTY NINE"):: RETURN
2030 CALL SAY("SEVENTY"):: RETURN
```

Slot Machine

If you have a streak of gambler in you, you'll like this program. It turns your TI into a Las Vegas slot machine. You start out with \$25, but don't be surprised at how quickly it disappears. This program uses the combinations found on a typical slot machine, and the payoff combinations are displayed on the screen. Good luck!

How Slot Machine Works

Line(s)

- | | |
|---------|--|
| 100 | Clear the screen and seed the random number generator. |
| 110 | Set the screen color to black. |
| 120-200 | Define the character patterns used in the programs. Images on the wheel will be double-sized unmagnified sprites. |
| 210 | Set the character pattern colors. |
| 220 | Read the ASCII code and color code for the seven possible patterns into PAT and COL arrays. |
| 230-250 | Read the combinations for each wheel of the slot machine into the W array. Each wheel consists of 19 patterns (oranges, lemons, bars, etc.). The actual information read into the array is a number which can be used as a subscript for the PAT and COL arrays. |
| 260-280 | Read the winning combinations (groups of three) into the PAY array. |
| 290 | Read the payoff odds into the ODDS array for each of the winning combinations. |
| 300-470 | Draw the slot machine, payoff board, and tote board. |
| 480 | Initialize tote board variables. If you want to start out with more than \$25 dollars (or coins), you change the ONHAND variable to the desired amount. |
| 490-510 | Set the initial patterns on the three slot machine wheels. |
| 520-540 | Display the current tote board status. |

Putting It All Together

- 550-590 Check for press of the space bar to start the wheels in motion. Line 590 picks a random starting position for each of the three wheels.
- 600-720 Put the wheels in motion. The next figure in each wheel is displayed in a rolling fashion. When any wheel reaches 19, it is wrapped around to 1.
- 730 Invoke routine to see if you won.
- 740 Go back and do it all again.
- 750-880 Determine winnings, if any. Display won or lost messages and update the appropriate counters. If the amount on-hand (line 870) reaches 0, then the game is over.
- 890-910 End of game.
- 920-970 DATA statements containing wheel information. Line 920 identifies the ASCII code numbers and color codes for the seven figures appearing on the wheels. Lines 930-950 identify the 19 figures on each of the three wheels. Line 960 identifies the winning combinations, and line 970 identifies the payoff odds for each winning combination.

Program 7-7. Slot Machine

```
100 CALL CLEAR :: RANDOMIZE
110 CALL SCREEN(2)
120 DIM W(3,19),PAY(10,3),ODDS(10)
130 CALL CHAR(40,"3C3C3C3C3C3C3C3C"):: CALL CHAR(9
6,"FFFFFFFFFFFFFFFF")
140 CALL CHAR(100,"00000000FF91A2C48891A2FF00000000
00000000FF112345891123FF0000000000")
150 CALL CHAR(104,"0003071F3F7FFFFFFFFF7F3F1F07030
000C0E0F8FCFEFFFFFFFFFEFCF8E0C000")
160 CALL CHAR(108,"00000000FF8EB58DB4B58DFF00000000
0000000000FF63ADAD23ABADFF0000000000")
170 CALL CHAR(112,"03070F3F3F7FFFFFFFFF7F3F3F0F070
3C0E0F0FCFCFEFFFFFFFFFEFCFCF0E0C0")
180 CALL CHAR(116,"01030509113078FCFC7831030707030
180402010088C9EBFBF9E8CC0E0E0C080")
190 CALL CHAR(120,"010103070F1F1F1F3F3F3F3F3F7F7FF
F8080C0E0F0F8F8F8FCFCFCFCFCFEFEFF")
200 CALL CHAR(124,"000001071F7FFFFF7F1F070100000000
00080C0F0FCFFFFFFFFFCF0C0800000000")
210 CALL COLOR(2,16,1,3,16,1,4,16,1,5,16,1,6,16,1,
7,16,1,8,16,1,9,15,1,11,7,1)
```

Putting It All Together

```
220 FOR L=1 TO 7 :: READ PAT(L),COL(L):: NEXT L
230 FOR L=1 TO 3 :: FOR L2=1 TO 19
240 READ W(L,L2)
250 NEXT L2 :: NEXT L
260 FOR L=1 TO 10 :: FOR L2=1 TO 3
270 READ PAY(L,L2)
280 NEXT L2 :: NEXT L
290 FOR L=1 TO 10 :: READ ODDS(L):: NEXT L
300 CALL VCHAR(1,14,40,24)
310 FOR L=1 TO 7
320 CALL HCHAR(L,16,96,14)
330 NEXT L
340 FOR L=3 TO 4
350 CALL HCHAR(L,18,32,2):: CALL HCHAR(L,22,32,2):
: CALL HCHAR(L,26,32,2)
360 NEXT L
370 CALL MAGNIFY(3)
380 FOR L=1 TO 8 :: FOR L2=1 TO 3
390 CALL SPRITE(#L2+((L-1)*3),PAT(PAY(L,L2)),COL(P
AY(L,L2)),16+(L*16),8+(L2*16))
400 NEXT L2
410 DISPLAY AT(4+(L*2),9):USING "###":ODDS(L);
420 NEXT L
430 DISPLAY AT(21,2):CHR$(116);CHR$(118);CHR$(116)
;CHR$(118);
440 DISPLAY AT(22,2):CHR$(117);CHR$(119);CHR$(117)
;CHR$(119);"{5 SPACES}5";
450 DISPLAY AT(23,2):CHR$(116);CHR$(118);
460 DISPLAY AT(24,2):CHR$(117);CHR$(119);"
{7 SPACES}2";
470 DISPLAY AT(12,15):"ON-HAND";:: DISPLAY AT(14,1
5):"WAGERED";:: DISPLAY AT(16,15):"WON";
480 WON=0 :: WAGERED=0 :: ONHAND=25
490 FOR L=1 TO 3 :: SEL=1+INT(RND*7)
500 CALL SPRITE(#25+L,PAT(SEL),COL(SEL),17,105+(L*
32))
510 NEXT L
520 DISPLAY AT(2,2):"***PAYOFF***";
530 DISPLAY AT(10,15):"*****MONEY*****";:: DISPLAY A
T(12,23):USING "#####":ONHAND;:: DISPLAY AT(14
,23):USING "#####":WAGERED;
540 DISPLAY AT(16,23):USING "#####":WON;
550 DISPLAY AT(22,14):"PRESS SPACEBAR";
560 CALL KEY(3,K,S):: IF K<>32 THEN 560
570 CALL HCHAR(22,16,32,14)
580 ONHAND=ONHAND-1 :: WAGERED=WAGERED+1
590 W1=1+INT(RND*19):: W2=1+INT(RND*19):: W3=1+INT
(RND*19)
```

Putting It All Together

```
600 FOR L=1 TO 40
610 IF L>24 THEN 650
620 W1=W1+1 :: IF W1>19 THEN W1=1
630 CALL SPRITE(#26,PAT(W(1,W1)),COL(W(1,W1)),17,1
  37)
640 CALL SOUND(-50,500,7,-7,0)
650 IF L>33 THEN 690
660 W2=W2+1 :: IF W2>19 THEN W2=1
670 CALL SPRITE(#27,PAT(W(2,W2)),COL(W(2,W2)),17,1
  69)
680 CALL SOUND(-50,500,7,-7,0)
690 W3=W3+1 :: IF W3>19 THEN W3=1
700 CALL SPRITE(#28,PAT(W(3,W3)),COL(W(3,W3)),17,2
  01)
710 CALL SOUND(-50,500,7,-7,0)
720 NEXT L
730 GOSUB 750
740 GOTO 530
750 WINNINGS=0
760 IF W(1,W1)=5 THEN IF W(2,W2)=5 THEN WINNINGS=5
  ELSE WINNINGS=2
770 FOR L=1 TO 8
780 IF W(1,W1)=PAY(L,1)AND W(2,W2)=PAY(L,2)AND W(3
  ,W3)=PAY(L,3)THEN WINNINGS=ODDS(L)
790 NEXT L
800 IF WINNINGS=0 THEN 850
810 DISPLAY AT(22,15): "*** WINNER ***";
820 CALL SOUND(500,800,3,1000,5):: CALL SOUND(1500
  ,-6,0)
830 WON=WON+WINNINGS :: ONHAND=ONHAND+WINNINGS
840 RETURN
850 DISPLAY AT(22,14): "*** YOU LOSE ***";
860 FOR L=1 TO 500 :: NEXT L
870 IF ONHAND=0 THEN 890
880 RETURN
890 DISPLAY AT(22,15): "MONEY GONE" :: DISPLAY AT(2
  4,15): "GAME OVER"
900 DISPLAY AT(12,23): USING "#####":ONHAND;:: DISP
  LAY AT(14,23): USING "#####":WAGERED;:: DISPLAY
  AT(16,23): USING "#####":WON;
910 GOTO 910
920 DATA 100,3,104,14,108,16,112,10,116,7,120,5,12
  4,12
930 DATA 1,2,3,4,5,4,3,2,3,6,2,4,5,4,3,2,3,4,5
940 DATA 1,6,2,6,5,6,5,6,3,6,3,6,5,6,5,4,5,6,3
950 DATA 1,7,4,7,2,7,4,7,4,7,2,7,6,7,4,7,2,7,4
960 DATA 1,1,1,3,3,1,6,6,6,6,6,1,2,2,2,2,2,1,4,4,4
  ,4,4,1,5,5,0,5,0,0
970 DATA 200,100,18,18,14,14,10,10,5,2
```

Index

- "Addition Climber"
 program 198-205
- "Air Defense" program 95-97
- allophone 161, 165, 168
 table 162-63
- "Allophone Number Lister"
 program 169
- "Allophone Stringer"
 program 171-72
- "Alphabet Invasion"
 program 184-90
- "American Siren" sound effect 145
- ASCII 14, 18, 32
 character code table 15
 character sets 25-26
 defined 9
- Assembler Language 10
- "Bach Prelude" program 138-43
- background color 26
- "Banzai Bunny" program 190-95
- bass 130-31
- "Bells" sound effect 146
- "Birds at Night" program 76-77
- bitmap mode 10
- "Blinky" program 35-37
- "Bomb and Explosion" sound
 effect 145-46
- CALL CHAR subprogram 18-19
- CALL CHARPAT subprogram 18
- CALL CLEAR subprogram 19
- CALL COINC subprogram 87,
 93-95, 102, 108
- CALL COLOR subprogram 25, 26,
 66
- CALL DELSPRITE subprogram 66
- CALL DISTANCE subprogram 87,
 97-99, 102
- CALL GCHAR subprogram 108, 109
- CALL HCHAR subprogram 27, 32,
 33
- CALL INIT subprogram 164
- CALL JOYST subprogram 71-72
- CALL KEY subprogram 72-75
- CALL LINK subprogram 164
- CALL LOAD subprogram 164
- CALL LOCATE subprogram 69-70
- CALL MAGNIFY subprogram 62-65
- CALL MOTION subprogram 70-71,
 75, 89
- CALL PATTERN subprogram 76
- CALL POSITION subprogram 87-89,
 93, 102, 108
- CALL SAY subprogram 153, 154
- CALL SCREEN subprogram 26, 52
- CALL SOUND subprogram 115-17,
 128, 131, 145, 146
- CALL SPGET subprogram
 153, 154
- CALL SPRITE subprogram 56-60
 example programs 60-65
- CALL VCHAR subprogram 27, 32
- character codes 15, 73
 changing 18
- character concatenation 33
- character definition 14-27
- character grid 13, 14, 16-18
- character set 25-26
- CHR function 19
- color codes
 table 25
- color monitor 6
- "COMPUTE! Cat" program 22-23
- "Computer" sound effect 144
- custom characters 18-27
- diagonals 30-33
- disk controller card 6
- disk drive 6
- DISPLAY AT statement 19, 20,
 24-25, 32, 33
- display mapping 54-56
- display planes 52-53
- "Dot Gobbler" program 77-80
- "European Siren" sound effect 145
- expression marks, musical 133
- Extended BASIC 3, 5, 149
- foreground color 26
- frequency (sound) 115
- graphics mode 10
- "Graph" program 43-48
- hexadecimal notation 16-17
- high-resolution graphics 40-48
 BASIC and 40
- "Histogram" program 28-30
- inflection codes 170
- joystick 6, 71-72
- "Kaleidoscope" program 67-69
- key-unit 73
- linear predictive coding (speech
 synthesis) 149
- "Meteors" program 99-102

- "Mimic" program 175-79
- "Morse Code" sound effect 144
- "Mouse Maze" program 103-12
- movement, sprite 51, 69-75
- multicharacter shapes 20-22
- multicolor mode 10
- "Music Demo 1" program 128-30
- "Music Demo 2" program 131-33
- "Music Demo 3" program 134-38
- music theory 117-23
- noise 116
- "Note Tutor" program 123-25
- number sign, CALL SAY and 154
- "Octaves" program 123
- pattern color table 54
- pattern generator table 54
- pattern identifier 18
- pattern name table 54
- Peripheral Expansion Box 6
- phoneme 161
- PHP-1500 speech synthesizer (*see* speech synthesizer)
- pitch 165
 - frequency chart 166
- pixel 13, 40, 51
 - defined 9
- "Plane" program 33-35
- plus sign, CALL SAY and 154
- resolution, defined 9
- "Rocket" sound effect 144
- "Shooting Gallery" program 179-84
- slope 165
- "Slot Machine" program 205-8
- sound
 - defined 115
 - examples 117
- sound effects 143-46
- Speak and Spell* products 149
- speech synthesis 5, 149-72
- speech synthesizer 149, 153
- split-keyboard mode 73-74
- sprite attribute table 54
- "Sprite Chase" program 91-93
- sprite collisions 87-112
 - speed and 102-3
- sprite color 66-67
- "Sprite Editor" program 80-83
- sprite generator table 54
- sprites 4, 51-83
 - advantages of 51
 - bitmap mode and 10
 - changing 76
 - deleting 66
 - large 61-66
 - moving 69-75
- "Talker" program 167
- "Tank Attack" program 37-40
- tempo, programming 127
- text mode 10
- text-to-speech diskette 6, 149, 161-72
- "3-D Shapes" program 41-42
- TI BASIC 3
- TI Extended BASIC 3
- TI screen 10-14
- TI text-to-speech diskette software (*see* text-to-speech diskette)
- TMS5200 speech synthesis chip 149
- TMS9918A video display processor (*see* VDP)
- tolerance, sprite collision and 93-94
- transparency 52
- VDP 52-54
 - display planes and 52-53
- VDP RAM 54
- vibrational pattern (sound) 115
- vocabulary speech synthesizer 149
 - expanding 155
 - table 150-53
- volume 115, 116
- waveform 115
- "Word Maker" program 156-61
- "Zone Defender" program 195-98

Notes

Notes



COMPUTE! Books

P.O. Box 5406 Greensboro, NC 27403

Ask your retailer for these **COMPUTE! Books**. If he or she has sold out, order directly from **COMPUTE!**

For Fastest Service
Call Our **TOLL FREE US Order Line**

800-334-0868
In NC call 919-275-9809

| Quantity | Title | Price | Total |
|----------|--|-----------------|-------|
| _____ | Machine Language for Beginners | \$14.95* | _____ |
| _____ | Home Energy Applications | \$14.95* | _____ |
| _____ | COMPUTE!'s First Book of VIC | \$12.95* | _____ |
| _____ | COMPUTE!'s Second Book of VIC | \$12.95* | _____ |
| _____ | COMPUTE!'s First Book of VIC Games | \$12.95* | _____ |
| _____ | COMPUTE!'s First Book of 64 | \$12.95* | _____ |
| _____ | COMPUTE!'s First Book of Atari | \$12.95* | _____ |
| _____ | COMPUTE!'s Second Book of Atari | \$12.95* | _____ |
| _____ | COMPUTE!'s First Book of Atari Graphics | \$12.95* | _____ |
| _____ | COMPUTE!'s First Book of Atari Games | \$12.95* | _____ |
| _____ | Mapping The Atari | \$14.95* | _____ |
| _____ | Inside Atari DOS | \$19.95* | _____ |
| _____ | The Atari BASIC Sourcebook | \$12.95* | _____ |
| _____ | Programmer's Reference Guide for TI-99/4A | \$14.95* | _____ |
| _____ | COMPUTE!'s First Book of TI Games | \$12.95* | _____ |
| _____ | Every Kid's First Book of Robots and Computers | \$ 4.95† | _____ |
| _____ | The Beginner's Guide to Buying A Personal Computer | \$ 3.95† | _____ |

* Add \$2 shipping and handling. Outside US add \$5 air mail, \$2 surface mail

† Add \$1 shipping and handling. Outside US add \$5 air mail, \$2 surface mail

Please add shipping and handling for each book ordered.

Total enclosed or to be charged.

All orders must be prepaid (money order, check, or charge). All payments must be in US funds. NC residents add 4% sales tax.

☐ Payment enclosed Please charge my: ☐ VISA ☐ MasterCard
☐ American Express Acc't. No. _____ Expires ____/____

Name _____

Address _____

City _____

State _____

Zip _____

Country _____

Allow 4-5 weeks for delivery.

If you've enjoyed the articles in this book, you'll find the same style and quality in every monthly issue of **COMPUTE!** Magazine. Use this form to order your subscription to **COMPUTE!**.

For Fastest Service,
Call Our **Toll-Free** US Order Line
800-334-0868
In NC call **919-275-9809**

COMPUTE!

P.O. Box 5406
Greensboro, NC 27403

My Computer Is:

- ☐ Commodore 64 ☐ TI-99/4A ☐ Timex/Sinclair ☐ VIC-20 ☐ PET
☐ Radio Shack Color Computer ☐ Apple ☐ Atari ☐ Other _____
☐ Don't yet have one...

- ☐ \$24 One Year US Subscription
☐ \$45 Two Year US Subscription
☐ \$65 Three Year US Subscription

Subscription rates outside the US:

- ☐ \$30 Canada
☐ \$42 Europe, Australia, New Zealand/Air Delivery
☐ \$52 Middle East, North Africa, Central America/Air Mail
☐ \$72 Elsewhere/Air Mail
☐ \$30 International Surface Mail (lengthy, unreliable delivery)

Name _____

Address _____

City _____

State _____

Zip _____

Country _____

Payment must be in US Funds drawn on a US Bank; International Money Order, or charge card.

- ☐ Payment Enclosed
☐ MasterCard

- ☐ VISA
☐ American Express

Account No. _____

Expires _____

/

If you've enjoyed the articles in this book, you'll find the same style and quality in every monthly issue of **COMPUTE!'s Gazette** for Commodore.

For Fastest Service
Call Our **Toll-Free** US Order Line
800-334-0868
In NC call **919-275-9809**

COMPUTE!'s GAZETTE

P.O. Box 5406
Greensboro, NC 27403

My computer is:

☐ Commodore 64 ☐ VIC-20 ☐ Other _____

01

02

03

- ☐ \$20 One Year US Subscription
☐ \$36 Two Year US Subscription
☐ \$54 Three Year US Subscription

Subscription rates outside the US:

- ☐ \$25 Canada
☐ \$45 Air Mail Delivery
☐ \$25 International Surface Mail

Name _____

Address _____

City _____

State _____

Zip _____

Country _____

Payment must be in US Funds drawn on a US Bank, International Money Order, or charge card. Your subscription will begin with the next available issue. Please allow 4-6 weeks for delivery of first issue. Subscription prices subject to change at any time.

- ☐ Payment Enclosed ☐ VISA
☐ MasterCard ☐ American Express

Acct. No. _____

Expires _____

/

The *COMPUTE!'s Gazette* subscriber list is made available to carefully screened organizations with a product or service which may be of interest to our readers. If you prefer not to receive such mailings, please check this box ☐.

Are you excited by your TI's graphics? Dazzled by its diversity of sound? Left speechless by the idea of synthesized speech—and baffled by how it's all done?

Then you need *COMPUTE!'s Guide to TI Sound and Graphics*. Using dozens of examples and clear, nontechnical explanations, it introduces you to the tremendous sound and graphics capabilities of your TI home computer.

Here are just a few of the things you'll find in this book:

- Fast and exciting arcade-style games
- Challenging educational programs, including one that actually talks to your child
- Clear instructions on using sprite graphics
- A versatile sprite editor
- Easy-to-use guidelines for creating custom sound effects
- Dozens of valuable graphics and sound subroutines
- Impressive musical subroutines and demonstrations
- A sophisticated speech program that uses TI's speech synthesis module and text-to-speech software to speak virtually any word that you type in

Whether you are a beginning TI user or an experienced programmer, *COMPUTE!'s Guide to TI Sound and Graphics* is a book you'll refer to again and again.